April 2008

Flight Attitude and Acceleration Data Recorder

Joshua Mark Lane
Worcester Polytechnic Institute

Michael John Roberts
Worcester Polytechnic Institute

Project ID: MQP FJL ACCE

Accelerometer Data Recorder

Michael Roberts
mroberts@wpi.edu

Joshua Lane
lanej@wpi.edu

Date: April 24, 2008

Professor Fred. J. Looft, Major Advisor
Professor and Department Head
Electrical and Computer Engineering
Worcester Polytechnic Institute

A Major Qualifying Project
submitted to the Faculty
of
WORCESTER POLYTECHNIC INSTITUTE
in fulfillment of the requirements for the
Degree of Bachelor of Science

Abstract

This document outlines the design and implementation of a low-cost, low-power acceleration data recorder for utility category gliders. Working from literature and experience, we have built a recorder that can accurately measure the forces and angle experienced by a glider in flight. The recorder is able to record and display acceleration force and angular rates of turn as well as display and analyze post-flight data utilizing software created for a personal digital assistant and personal computer.

Table of Contents

Abstract 2
Table of Contents 3
Table of Figures 6
1 Introduction 8
1.1 Project Statement 11
1.2 Report Organization 11
1.3 Summary 12
2 Background 13
2.0 Introduction 13
2.1 Basic Operation of a Glider 13
2.2 G-Force 14
2.3 Accelerometers 15
2.3.1 Types of Accelerometers 15
2.3.2 FAADR Accelerometer Properties 17
2.4 Electric Compass 18
2.5 Attitude Indicators 18
2.6 Rate Sensors 20
2.7 Serial Communication 21
2.8 Summary 22
3 Problem Statement 24
3.1 Introduction 24
3.2 Project Statement 24
3.3 Requirements 24
3.4 Desirements 25
3.5 Summary 25
4 System Design 26
4.1 Introduction 26
4.2 Overall System Design 26
4.2.1 Enclosure 27
4.2.2 Electric Compass 27

4.2.3 Accelerometer 28
4.2.4 LCD Module 28
4.2.5 Removable Media Storage Module 29
4.2.6 Microprocessor Unit 29
4.3 Component Selection 30
4.3.1 Electric Compass 30
4.3.2 Accelerometer 33
4.3.3 LCD Module 37
4.3.4 Microprocessor Unit 37
4.3.5 Removable Media Storage Module 38
4.4 Summary 38
5 Results 39
5.1 Introduction 39
5.2 Hardware Implementation 39
5.2.1 MSP430F1491 Microcontroller 39
5.2.2 Crystalfontz Graphic LCD 40
5.2.3 DOSonCHIP-SD Module 42
5.2.4 SCA3000-E04 3-Axis Accelerometer 44
5.2.4 Angular Rate Sensor MLX90609 46
5.3 Software Implementation 48
5.3.1 Accelerometer 48
5.3.2 Rate Sensor 51
5.3.3 LCD Display 53
5.3.4 SD Card 55
5.3.5 Main 57
5.3.6 Analysis Software 57
5.4 Analysis 61
5.4.1 Requirements 61
5.4.2 Issues 63
5.5 Summary 64
6 Conclusion 65
6.1 Project Summary 65
6.2 Future Work 66
Works Cited 68
Appendix A: Source Code 70

Accelerometer Header 70
Accelerometer Source 70
Rate Sensor Header 74
Rate Sensor Source 75
SD Card Header 76
SD Card Source 78
LCD Header 83
LCD Source 85
T6963 (LCD Processor) header 97
T6963 (LCD Processor) Source 99
Sonic Alert Header 104
Sonic Alert Source 104
Appendix B – Design Schematics 106
Sensor Board 106
Main Board 107
Appendix C – PCB Board Design 108
Sensor Board 108
Main Board 110
Appendix D – Horizon Sprite and Logic Generator Source Code 111
Appendix E - Analysis Software Source Code 112
Main Analysis Class 112
Data Set Handler Class 116
Data Parser 118
Data Object 119
Initial Screen XML 120
Final Screen XML 120

Table of Figures

Figure 1: An Artificial Horizon [16] 8
Figure 2: The three axes of an aircraft [14] 9
Figure 3: AHRS-2 Display of ARS-2 Attitude Reference System from MGL Avionics (Not shown: AV-2 Universal Display Indicator) 10
Figure 4: Demonstration of Air Flow Generating Lift [17] 13
Figure 5: Resulting Motion from Elevators [10] 14
Figure 6: A capacitive accelerometer with no forces acting upon the center plate. [19] 16
Figure 7: A capacitive accelerometer with a force being applied to the center plate. [19] 17
Figure 8: Melexis MLX90609 angular rate sensor. (Right) Single capacitive element. (Left) Internal structure. [12] 21
Figure 9: SPI Master and SPI Slave device in 4-wire mode 22
Figure 10: SPI Cooperative Bus or Daisy Chained Configuration 22
Figure 11: FAADR System Block Diagram 26
Figure 12: Enclosure Location and Inventory 27
Figure 13: Electric Compass System Block Diagram 27
Figure 14: Accelerometer System Block Diagram 28
Figure 15: LCD Module System Block Diagram 29
Figure 16: Removable Storage Media Module System Block Diagram 29
Figure 17: Microprocessor Unit System Block Diagram 30
Figure 18: Chart of Electric Compass Unit Scores 32
Figure 19: Chart of Weighted Scores of Electric Compass Units 33
Figure 20: The V2Xe 2 Axis Digital Compass (http://www.pnicorp.com) 33
Figure 21: Chart of Considered Accelerometers Unit Scores 35
Figure 22: Chart of Considered Accelerometers Total Scores 36
Figure 23: SCA3000-E04 3-Axis Accelerometer 36
Figure 24: CFAG128128A-STI-TZ from CrystalFontz America Inc 37
Figure 25: B1491 Breakout Board for the MSP430F1491 38
Figure 26: Breakout Board for the DOSonCHIP FAT16/FAT32 Module 38
Figure 27: Schematic Diagram of MSP430 to LCD Display Interface 41
Figure 28: Schematic diagram of DOSonChip and MSP430 Microcontroller interconnection 43
Figure 29: MSP430 connections to Ethernet port which correspond to accelerometer connections 44
Figure 30: Accelerometer connections to Ethernet port which correspond to MSP430 connections 45
Figure 31: The V2Xe 2 Axis Digital Compass (http://www.pnicorp.com) 46
Figure 32: AOS EZ-Compass 3 (www.aositilt.com) 47
Figure 33: The angular rate sensor MLX90609 (http://robosavvy.com) 48
Figure 34: Accelerometer Initialization Code 49
Figure 35: Accelerometer Software Diagram 50

Figure 36: Bit Structure of the SCA3000-E04 16-bit Registers 50
Figure 37: 16-bit Register Read from the SCA3000-E04 51
Figure 38: Rate Sensor Initialization Code 52
Figure 39: Rate Sensor Software Diagram 53
Figure 40: LCD Display Software Diagram 54
Figure 41: FAADR Active LCD Display 55
Figure 42: SD Card Software Diagram 56
Figure 43: Internal Software System Block Diagram 57
Figure 44: The initial screen of the analysis software 58
Figure 45: Software Analysis initial screen with populated values 59
Figure 46: The final screen displayed by the analysis software which allows the user to view 5 different graphs generated from the given data 59
Figure 47: Graph from analysis software showing pitch acceleration over time 60
Figure 48: Graph from analysis software showing total acceleration over time 60
Figure 49: Graph showing the Yaw acceleration over time 61
Figure 50: Graph showing the total times over 6Gs for total, pitch, roll and yaw acceleration 61
Figure 51: FAADR Main Enclosure with Accessory Power Socket Adapter and Ethernet cable 65
Figure 52: FAADR Sensor Enclosure with Ethernet cable 66
Figure 53: FAADR Final Product 66

1 Introduction

All aircraft in the United States are required by Federal Aviation Administration (FAA) regulations to have three instruments; an altimeter, an airspeed indicator, and a magnetic direction indicator [8]. However, pilots often use more than the required instruments to help them fly. One of these instruments is an artificial horizon, shown in Figure 1.

Figure 1: An Artificial Horizon [16]

A pilot uses an artificial horizon to assess the pitch and roll of the aircraft. As seen in Figure 1, the instrument is designed to look similar to a natural horizon. The middle indicator indicates the attitude of the airplane and the markings indicate specific increments of pitch. The marking on the outside indicate increments of roll. As shown Figure 2, the roll specifies to what extent the airplane is banked left or right. In other terms, roll is defined as the extent an aircraft is rotated around the longitudinal axis (from the nose to the tail of the aircraft).

The pitch of the aircraft specifies the degree which the nose of the aircraft is directed up or down. Pitch can also be described as the degree which the aircraft is rotated around the lateral axis (extends for one wingtip to the other, see Figure 2). If the roll and pitch angles are zero then the wings and nose of the aircraft are both parallel to the horizon and the pilot experiences straight and level flight.

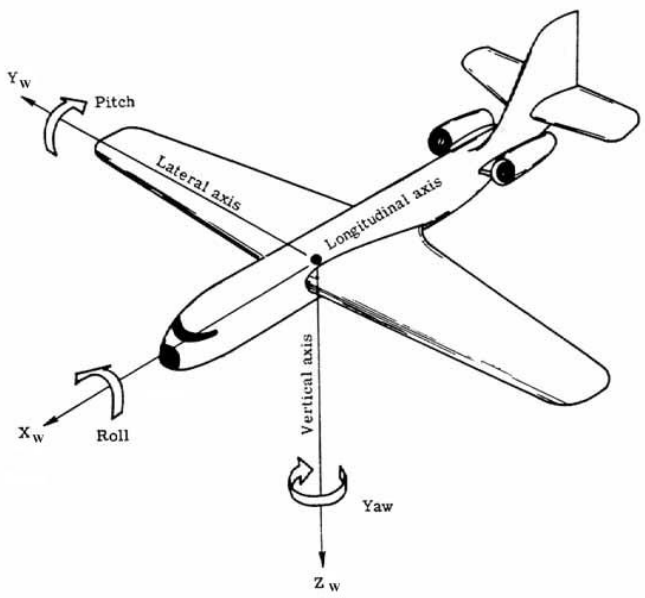


Figure 2: The three axes of an aircraft [14]

From a pilot's perspective, pitch and roll knowledge are essential information because they tell the pilot the aircraft attitude relative to straight and level flight. Although not a required instrument on all aircraft, artificial horizons are most useful when the natural horizon is not visible. This situation can occur, for example, when a pilot is in a cloud. In the United States, unpowered aircraft pilots are prohibited from flying inside clouds but such situations can occur [8]. When Visual Flight Rules (VFR) no longer applies, such as an aircraft inside a cloud, the pilot is in Instrument Flight Rules (IFR) conditions. Pitch and roll data is critically important in such a situation since, for example, if a pilot banks the aircraft, the airspeed of the aircraft will decrease because the drag on the aircraft increases [16]. Similarly, pitch has an impact on the rate of change of altitude.

When flying through a cloud, a pilot would also need to refer to a magnetic direction indicator (compass) to assess the direction of the aircraft. A compass is an instrument used to measure heading based on the device's deviation from the Earth's magnetic north pole.

The Federal Aviation Administration categorizes all aircraft that fly in the United States. These categories are determined by many factors. One of the most important factors is the amount of loading expected. The load factor is the ratio of total lift generated by an aircraft to the total weight of the aircraft. Other factors include use, type of construction, propulsion system (if any), electrical system, and aerobatic capabilities (if any) that the aircraft can withstand before structural failure. For example, the most common categories for light aircraft structural loading are Normal, Utility, and Aerobatic.

A glider is an a lightweight aircraft that is included into the FAA's light aircraft categories and stays aloft primarily by gaining altitude using columns of rising air. Glider pilots, like powered aircraft pilots, are concerned about the level of stress placed on the gilder. There are many ways that a pilot can create excessive stress on a glider. One of the most common sources is extreme maneuvering. By combining

extreme pitch, roll and yaw changes a pilot can rapidly alter the orientation of the glider and possibly over stress the aircraft.

If a glider is stressed beyond the designed tolerance range for the glider, the pilot may wish to have the glider inspected. There are many different types of instruments that a pilot can buy that can record the stresses experienced by an aircraft during flight. However, these instruments, such as the ARS-2 Attitude Reference System from MGL Avionics shown in Figure 3, at $869.00 may stress the owner's wallet and often require more power than is available [16].

Figure 3: AHRS-2 Display of ARS-2 Attitude Reference System from MGL Avionics (Not shown: AV-2 Universal Display Indicator)

How can a pilot have access to this crucial information without spending an excessive amount of money and quickly exhausting the glider's power supply?

1.1 Project Statement

The purpose of this project was to use electrical and computer engineering concepts to design and build a Flight Attitude and Acceleration Data Recorder, referred to hereafter as FAADR. The FAADR will record the three-dimensional acceleration vector of a utility category glider and display average, current and maximum data points on a graphical Liquid Crystal Display (LCD). The FAADR will use a Secure Data (SD) card to record data on in order to make the results portable and easily accessed. The FAADR will include software to display and process the data on a Personal Digital Assistant (PDA) and Personal Computer (PC).

The FAADR will be able to:

- Make tri-axial measurements of upwards of ±5 g with an accuracy of ±0.1 g
- display acceleration axis values, both peak, average and instantaneous
- reset functionality of average and maximum values
- attitude indicator (artificial horizon) with at least 2 degrees accuracy

1.2 Report Organization

The Section 2 Background of this report will illustrate details of the FAADR by describing what exactly is being measured and how it is quantified. This section will also provide information on basic concepts of flight and explain how specific portions of the FAADR work.

Section 3 Problem Statement will demonstrate details of the FAADR including:

- specific goals
- requirements
- functional specifications

Section 4 System Design outlines key components within the FAADR, how they will interact with other components in software and hardware and why the particular component was chosen. Section

5 Results, will explain how each component was physically implemented, the code that makes each component work, and how well the FAADR met the requirements and goals defined in Section 3 Problem Statement.

Other information about the project including schematics, PCB board designs, and the code for the analysis software and the microcontroller operation can be found in the appendices.

1.3 Summary

The FAADR will give glider pilots the ability to monitor the stress their glider endures during flight without consuming undue power and money. By illustrating the details of a flight, the instrument will be an affordable method of improving flight safety by better informing pilots of the aircraft's current condition.

2 Background

2.0 Introduction

The purpose of this chapter is to provide a detailed description of the background FAADR material that is required to understand the concepts of our Accelerometer Data Recorder project. This chapter will introduce ideas including a brief synopsis of how a glider operates, the forces a glider encounters during flight and instruments used by pilots which are relevant to the project. This information should demonstrate to the reader how the data recorder will operate and why it is a feasible and important undertaking.

2.1 Basic Operation of a Glider

A glider is an aircraft which flies through the air without any form of mechanical propulsion. A glider uses vertically moving air currents and forward flight in order to reach a sufficient flying altitude and to generate enough lift in order to sustain flight. Vertically moving currents are utilized by glider aircraft because the aircraft are lightweight enough for the air current to exert enough force so that the glider gains altitude [7]. Although a glider gains altitude using these air currents, it is always moving downward with respect to the local air.

A glider must maintain an air speed which is fast enough for the wings to generate enough lift so that the craft does not stall [7]. The speed of the glider in relation to the air that is moving past it is called airspeed which is essential for a glider to stay aloft. As seen in Figure 4, the airspeed and the angle of attack are the primary factors in how much lift a glider will generate. When a glider stalls, it is not generating enough lift to stay aloft and will begin falling toward the ground.

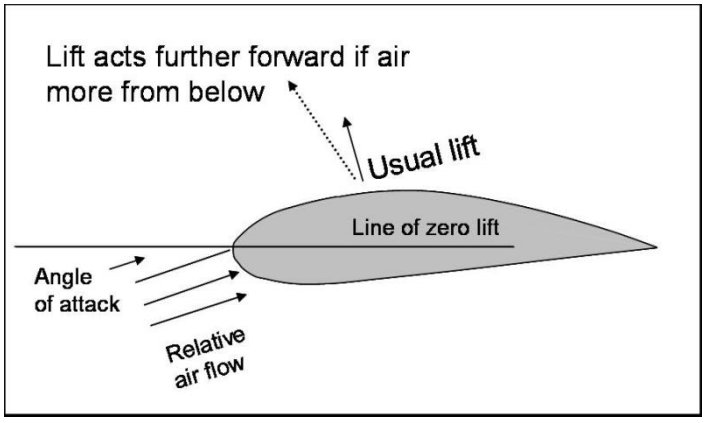


Figure 4: Demonstration of Air Flow Generating Lift [17]

Since a glider has no form of propulsion, the pilot of a glider needs to use pitch in order to maintain, reduce or increase airspeed [3]. Shown in Figure 2, pitch, which is defined as the rotation around the lateral axis, can be identified by the nose of the plane in relation to the horizon. When the nose of the aircraft is directed above the horizon it will begin to lose airspeed and when the nose is pointed below

the horizon it will gain speed. This is because when the craft is moving upward it is resisting gravity and when it is moving downward it is aligning with the pull of gravity. As shown in Figure 5, a plane can either pitch up or pitch down by using elevators which can be found on the tail of the craft [3].

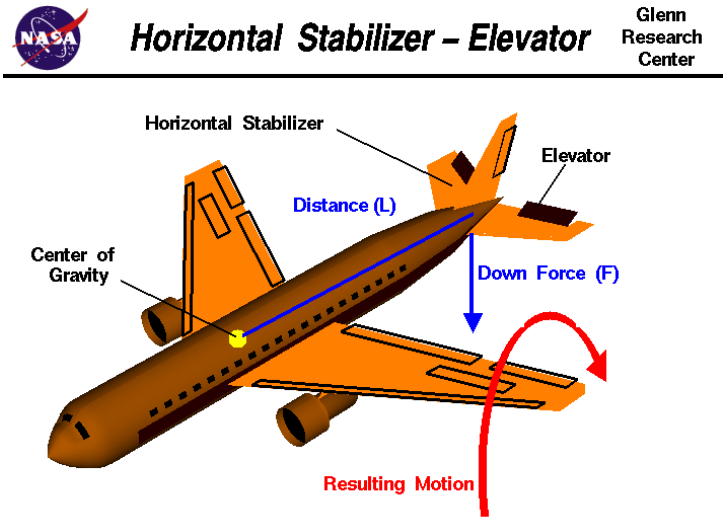


Figure 5: Resulting Motion from Elevators [10]

Another axis that gives the pilot control of the aircraft (shown in Figure 2) is called the roll or bank of the glider. A pilot can turn an aircraft by slightly rolling to one side and then increasing the lift by pulling back on the yoke [3]. This will make the glider turn onto a new heading because banking a plane causes a change in the lift vector since the wings are no longer level with the ground. The altered lift vector causes the glider to turn because the force exerted on the wings is no longer acting in only an upward direction, but also in the direction in which the wings are banked.

A glider is capable of these banking maneuvers due to ailerons, which are flaps on the wings of the craft that are controllable by the yoke [3]. To initiate a roll, the aileron on one wing will move up and the aileron on the other wing will move down thus creating more lift in the downward moving aileron and less in the upward moving aileron. The difference in lift will cause the plane to roll toward the side of lower lift.

Shown in Figure 2, the final axis of a glider is called the yaw axis of the craft. Yaw is the side to side movement of the aircraft which is primarily controlled by the rudders [3]. The rudder is a vertical flap on the tail of the plane which can force the nose of the aircraft to yaw in either direction. The force that is generated by the rudder moving from side to side translates to the direction of the nose of the plane.

2.2 G-Force

G-force (referred to as a “G” or loading) exerted upon aircraft is of particular importance to this project since one of the main functions of the proposed data recorder is to display and record the g-forces exerted upon a glider.

A G can be understood as the apparent weight of an object [4]. To understand what is meant by the term "apparent weight" one must first understand the idea of weight. The weight of an object which is completely still on earth, is determined by both its mass and gravity. Gravity is the acceleration which everything free falls to the earth. The force which is exerted on any still object (on the ground) is one G.

An aircraft can experience more than or less than one G since it can accelerate into any direction in three dimensional space. This leads to the fact that an aircraft can have g-force exerted on it from any direction. When a G is exerted upon an aircraft, it is known as loading and excessive loading can potential destroy an aircraft.

For example, an aircraft which is accelerating straight up into the air away from the ground at 9.81m/s^2, equivalent to gravity, would experience 2 Gs. This is because the craft is having both gravity and its own acceleration act on it in the same direction. Since the acceleration is equal to gravity, which itself is 1G, the addition of the acceleration and gravity yields 2 Gs.

G-force is important to this project and especially to pilots because a g-force of too great a magnitude (positive or negative) can cause an aircraft to structurally fail [4]. Of course, this of monumental concern to pilots because damaging a glider in mid-flight by over stressing the craft can be dangerous or even lethal. The FAADR will display the current Gs that the craft is experiencing in order to alert pilots of the current loading so that they do not exceed their craft's specifications.

2.3 Accelerometers

Accelerometers are sensors that are used to measure the amount of g-force that is being exerted upon them. This relates to a glider because the loading that is being exerted upon the craft is also being exerted upon any accelerometer that is contained inside the craft. Accelerometer data is priceless because a pilot can see a measurement of how much stress the aircraft is under in particular situations and maneuvers [4].

Accelerometers can vary widely between models in the following basic ways:

- The maximum level of constant g-force that the accelerometer can handle
- The maximum (impact) g-force that the accelerometer can handle
- The number of axes an accelerometer can measure (between one and three)

2.3.1 Types of Accelerometers

Accelerometers can vary in the aforementioned ways because accelerometers can be constructed in a many ways which fall under different categories including:

- Piezoelectric
- Potentiometric
- Reluctive
- Servo

- Strain Gauge
- Capacitive
- Vibrating Element [18]

This document will not detail all of the previously listed accelerometers, but will only thoroughly detail the capacitive accelerometer because that is the type used in the FAADR. For more information on the listed accelerometers visit: http://bits.me.berkeley.edu/beam/acc_2b.html

As shown in Figure 6, capacitive accelerometers use a capacitive divider with a common center plate in order to measure the acceleration force under gone by the sensor. The setup is created by using two separate outer plates which form a capacitor with a free moving center plate. When the sensor is at rest, the capacitances negate each other since the distances between the center plate and the outer plates are equal.

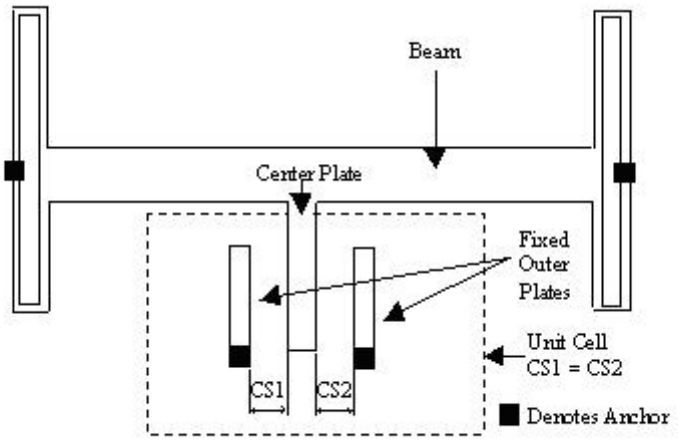


Figure 6: A capacitive accelerometer with no forces acting upon the center plate. [19]

Since the center plate of a capacitive accelerometer is free moving, applying Gs to the accelerometer (shown in Figure 7) will cause the center plate to move toward one of the outer plates and away from the other plate. This will cause the capacitances to become uneven causing a signal voltage to be applied to the center plate. The magnitude of this signal voltage changes depending on the distance the center plate moves and therefore, the magnitude of the signal voltage can be used as an accurate measure of how many Gs were applied to the sensor.

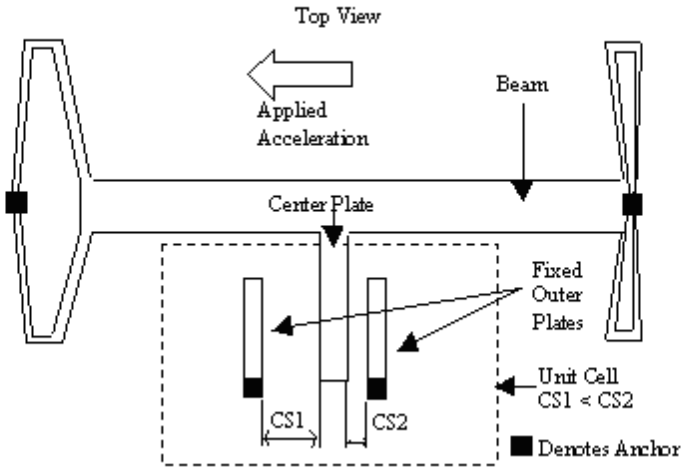


Figure 7: A capacitive accelerometer with a force being applied to the center plate. [19]

The capacitive accelerometer that has been described can only measure the Gs applied to the sensor along a single axis. A tri-axial accelerometer (required by the FAADR) can be constructed by mounting three capacitive accelerometers orthogonally and would therefore allow three signal voltages to represent the Gs applied to each axis.

2.3.2 FAADR Accelerometer Properties

The accelerometer properties that are essential to the FAADR's continued successful operation are listed in Section 2.3 Accelerometers. The listed properties are important because they detail how much force the FAADR can handle and also the amount of axes the FAADR can measure.

An accelerometer can be rated to sustain many different ranges of g-force. The amount of sustained g-force that the accelerometers need handle without breaking down for this project is approximately between positive and negative ten g-forces. This means that if a pilot has an aircraft that can sustain a five g-force bank or climb then the accelerometer should be able to adequately measure this g-force without risk of being damaged.

The impact g-force an accelerometer can experience without being damaged is not as pertinent to this project because if an aircraft experiences sudden jolts that are over the rating of the accelerometer the pilot would be much more concerned about the damage to his craft than to the FAADR. One of the only ways that this impact g-force rating could be exceeded is if the accelerometer is dropped an experiences a nearly instant deceleration when it hits the ground.

Another important difference between accelerometer models is that the number of axes that are measured can vary from one to three. This is pertinent to flying a glider because a pilot would not only want to see the g-force that is being exerted on his aircraft in only one direction. A pilot would want to see the force that is being exerted on the aircraft on each of its axes. An accelerometer that is housed in an aircraft would certainly need to measure all three axes, but three separate accelerometers have the

ability to measure all directions just as a tri-axial accelerometer. Despite the way that accelerometers can be setup in order to accurately measure the g-force exerted on a plane, it is paramount that all axes are considered.

The accelerometer data recorder project hopes to improve on existing flight accelerometer instruments is by allowing a pilot to purchase a relatively inexpensive device that can both display, record and play back the g-forces that were exerted during the flight of the craft.

2.4 Electric Compass

An electric compass is a device that can measure deviations from the Earth's magnetic north pole. Electric compasses that are manufactured by PNI use a principle known as magneto-inductance, in order to measure changes relative to the Earth's magnetic field.

A magneto-inductive sensor uses a single solenoid winding which serves as an inductor that oscillates at a predetermined frequency. A magnetic field element that is parallel to the solenoid axis can influence the inductance of the coil and therefore, the frequency of oscillation when the field changes 69[20]. By measuring the changes in oscillation an electric compass can effectively measure the angle which the coil or field was rotated.

A single magneto-inductive sensor can only measure one axis since it can only detect angular changes in the field element that is parallel to the coil axis. This fact shows that two coils will need to be used in the bi-axial electric compass that is needed for the FAADR project. By using the two coils in one instrument, two angles can be measured both independently and simultaneously.

This technology is important to pilots because a two axis electric compass has the capability of measuring the angle of bank and pitch. When a pilot rolls or pitches his craft up or down it will deviate from its original course and the electric compass will be able to show the angular change from straight and level flight.

The display of the attitude of the aircraft, the angle of both pitch and roll, is accomplished by instruments known as attitude indicators. These devices are important for all pilots who need to know their pitch and roll, as will be explained in detail in Section 2.5 Attitude Indicators.

2.5 Attitude Indicators

An attitude indicator is a crucial instrument for any pilot because it provides invaluable information about both the pitch and roll of the aircraft [1]. Figure 1 shows a typical attitude indicator that could be found in nearly all modern aircraft.

The most prominent portion of the attitude indicator is the mock aircraft that is found at the center of the instrument. It is the frame of reference with which a pilot can read the angle at which the craft is banking, climbing or descending [1]. This is essential because a pilot always needs to know the orientation of the aircraft so that he can maintain control of the flight.

Another component of the attitude indicator is the roll and pitch degree marks [1]. The pitch marks on an attitude indicator are often labeled in ten degree increments with an additional marking between each label so that the pilot is aware of every five degree change. The roll indicators also show that the plane is banking in five degree increments. This gives the pilot complete awareness as to the attitude of the aircraft and allows them to know information such as how much more bank is needed in order to reach a fifteen degree turning angle.

There are two types of attitude indicator that can be used to display the attitude of the aircraft. They work on inherently different principles as one type is purely mechanical in nature and the second type is electronic in nature. The different types include:

- Gyroscopic Attitude Indicator
- Electronic Attitude Indicator

The gyroscopic attitude indicator operates on a simple premise where the gyro stays in a fixed position and the glider moves around that position. This can be achieved by mounting the gyro on two gimbals (pitch and roll gimbals allow the gyro to rotate around an axis) so that the gyro can rotate freely while the glider is in flight [21]. As shown in Figure 1, a stationary artificial aircraft and angle indicator are usually mounted in front of the gyro so that the pilot can reference the stationary portions with respect to the gyro. Markings indicating pitch and roll angles are used by the pilot so that the pitch and roll are always known.

An electronic attitude indicator will be replicated in the FAADR by using an electric compass. As described in Section 2.4 Electric Compass, an electric compass can measure deviations from true magnetic north which allows for a measure, in degrees, of both pitch and roll of an aircraft. The attitude indicator can retrieve the necessary information about pitch can roll from the electric compass and project an artificial horizon on a graphic display.

The display will generate an image similar to Figure 1, by recreating a plane against a horizon with pitch and roll angular markings. When a change in angle is received from the electric compass the display will be refreshed so the plane reflects the change in angle against the artificial horizon.

The FAADR will perfectly simulate a gyroscopic attitude indicator, but will have certain advantages over the gyroscopic indicator which will include:

- Recording of the angular values
- FAADR horizon will not tumble and become essentially unusable at large angles of pitch and roll [22]

The only disadvantage the FAADR will have over the gyro horizon is that the gyro horizon is nearly perfectly accurate since it is a completely analog mechanical device. There is very little error in the gyroscopic horizon because, as long as it does not tumble, it will stay completely stationary while the plane moves giving the gyro near perfect accuracy. The accuracy of the electric compass used to measure the angles of pitch and roll is simply not as accurate because it has a maximum error of 2

degrees. This error, however, is acceptable because a 2 degree error is not large enough to cause pilot disorientation.

The FAADR will be invaluable to pilots because it will provide a cost effective alternative to other commercially available attitude indicators. The FAADR will also have the ability to record pitch and roll which will allow the user to view statistics and a recreated flight on a personal computer or personal digital assistant after landing the plane.

2.6 Rate Sensors

Rate sensors detect rotational acceleration around a vertical axis using two different piezoelectric or micromechanical elements. Both of these sensors operate on the Coriolis force, which is named after the French mathematician Gaspard Gustave Coriolis, who described it in the paper "Sur les equations du mouvement relatif des systèmes de corps" ("On the equations of relative motion of a system of bodies") [10]. The Coriolis force is experienced by moving objects in a rotating frame of reference and is mathematically dependent on the centrifugal force and the velocity of the moving object. Coriolis Acceleration can be derived from the expression shown in Equation 1. The expanded expression that shows the useful mathematical dependencies is shown in Equation 2 where a is the absolute acceleration, a_r is the observer's acceleration, $\omega \times (\omega \times r)$ is the centrifugal acceleration and $2\omega \times v_r$ is the Coriolis Acceleration.

$$\left(\frac{d\mathbf{B}}{dt}\right)_f = \left(\frac{d\mathbf{B}}{dt}\right)_r + \omega \times \mathbf{B}$$

Equation 1: Base Equation for the Derivation of the Coriolis Acceleration [10]

$$\mathbf{a} = \mathbf{a}_\mathbf{r} + 2\omega \times \mathbf{v}_\mathbf{r} + \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{r})$$

Equation 2: Expanded Expression of Coriolis Acceleration [10]

Rate sensors exploit the Coriolis force in order to determine rotational rate. Micromechanical sensors contain a capacitive acceleration sensor on an electronically-controlled oscillating structure. The acceleration can then be measured as the product of the yaw rate and the velocity of the oscillating structure. The figure below shows a capacitive element and oscillating structure of the Melexis MLX90609 angular rate sensor which is a micromechanical yaw rate sensor

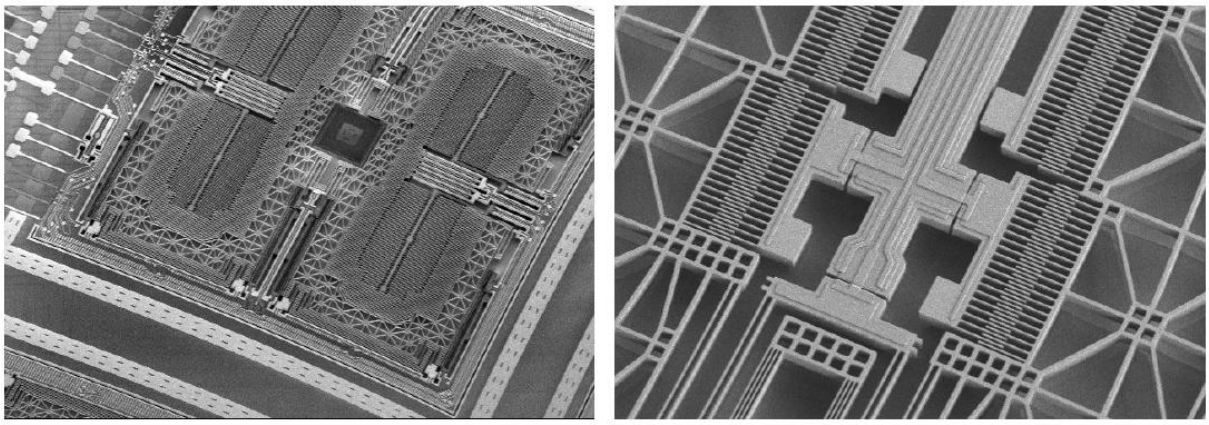


Figure 8: Melexis MLX90609 angular rate sensor. (Right) Single capacitive element. (Left) Internal structure. [12]

2.7 Serial Communication

In embedded systems, serial communication is the method of sending one bit at time, sequentially, over a bus or interconnection network. Serial communication contrasts to parallel communication where multiple bits are communicated in one clock cycle over two or more paths. There are many different types of serial communication protocols, but some of the common ones include Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter (UART), and RS-232.

SPI communication is common in integrated circuits (ICs) because it requires fewer pins and, therefore, less money to manufacture. Only three pins are needed to create a complete SPI port, Master In-Slave out (MISO), Master Out-Slave In (MOSI), and clock. Figure 9 shows a master and slave device in 4-wire mode which is identical to three wire mode with the addition of a chip select (CS) bit which is an enable signal for the slave device. The master SPI device is responsible for generating the clock signals that correspond with input and output to the slave device. A typical command-response procedure for SPI communication:

1. Master is requested to send a command
2. The chip select pin goes low for the appropriate device
3. Master generates the input and clock signals
4. Slave activates with the chip select signal going low
5. Slave buffers the input using the clock signals generated by the master
6. Master writes a query command asking the slave if it is done processing the first command by performing a “dummy-write” where the input to the slave is disregarded but the clock signals for the clocking output from the slave to the master as generated.
7. When the slave is ready to respond, master does a “dummy-write” to clock in the initially requested data.

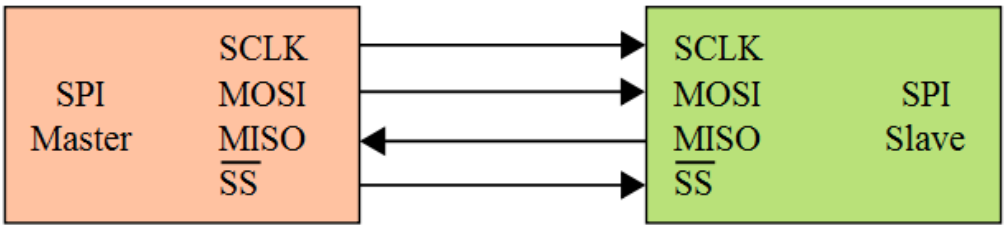


Figure 9: SPI Master and SPI Slave device in 4-wire mode

SPI devices can be formed into an independent bus by adding an additional chip select (CS) pin for each device in the chain. This allows many different SPI devices to be operated on the same SPI port. In the independent bus configuration, all SPI slave devices output pins are connected directly to the master.

In a "daisy-chain" or cooperative bus as shown in Figure 10, the input of the first SPI device is the output of the master, the output of that SPI device is the input to the next slave in the chain, and the last slave is the input to the master. With the addition of a multiplexer to either bus configuration, the chip select pin count can be reduced.

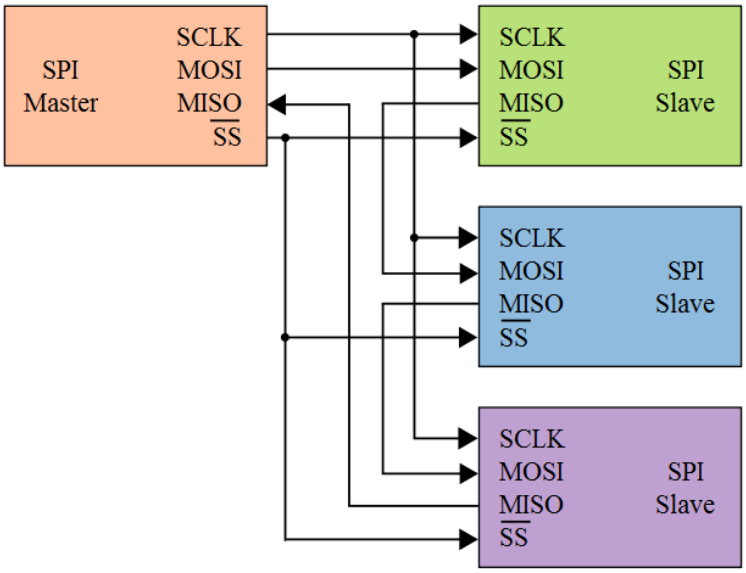


Figure 10: SPI Cooperative Bus or Daisy Chained Configuration

2.8 Summary

In this chapter we have described all necessary background material in order to understand the scope of the Accelerometer Data Recorder project and the flight mechanics the instrument will need to measure, display and record. The chapter conveyed that a glider has three axes, yaw, pitch and roll, on which the recorder will need to measure g-force. G-force is important to a pilot and his craft because over

stressing an aircraft with g-force can damage it. The concept of the Coriolis force is essential to measuring the pitch and roll of the aircraft. Serial communication is the main form of communication used in embedded systems in order to reduce the overall package footprint. Also shown in this chapter, was how the data recorder will function as an attitude indicator and display both the pitch and roll attitude to the pilot. Overall, this project is significant because it combines two crucial flight instruments and allows the user to record and then playback the data gathered during a flight.

3 Problem Statement

3.1 Introduction

The purpose of this section is to clearly define the requirements of the FAADR. This will allow a distinct measurement of the success of the FAADR.

3.2 Project Statement

The purpose of this project is to design and build a Flight Attitude and Acceleration Data Recorder, referred to hereafter as FAADR. The FAADR will record the three-dimensional acceleration vector of a Utility category glider and display average, current and maximum data points on a graphical Liquid Crystal Display (LCD). The FAADR will use a Secure Data (SD) card to record data on in order to make the results portable and easily accessed. The FAADR will include software to display and process the data on a Personal Digital Assistant (PDA) and Personal Computer (PC).

3.3 Requirements

The FAADR will be designed to meet specified requirements. Each will be listed below along with the justification for that requirement.

1. Require less than 2 W. The FAADR will be operated in a limited power environment where the main power supply is the 12V battery with an 8 Ah capacity on the glider.
2. Draw power from accessory power socket. This will allow the FAADR to be easily powered from an existing power connections found on most gliders.
3. Record tri-axial g-force measurements
    a. Range: +/-5G. Utility category gliders are rated to +4.4 and -1.76G.
    b. Accuracy: 0.1G. This is a reasonable resolution for G-force measurements.
4. Record heading
    a. Range: 360º. The FAADR will report and record heading in any direction.
    b. Accuracy: 2º. This is a reasonable resolution for heading measurements
5. Record up to 12 hours of data. This is the length of a very long flight in a glider. The following data points are considered necessary for reconstructing and analyzing a flight.
    a. Heading
    b. Tri-axial acceleration data
    c. Time
    d. Pitch
    e. Roll

6. Display data on graphical LCD. The FAADR is required to show the following data points on a LCD to provide useful data to the pilot because it is relevant information for the pilot.
    a. Maximum G
    b. Current G
    c. Pitch
    d. Roll
7. Display an artificial horizon. Using pitch and roll data the FAADR will display an artificial horizon in order for the data to be easily interpreted.
8. Display current heading. Considered to be relevant information for the pilot.
9. Reset display values. The FAADR will give the pilot the ability to reset data points computed and displayed during the flight.
    a. Maximum G
10. Include analysis software that runs a PDA and PC. This will allow the pilot to view and analyze data from the flight on different platforms.

3.4 Desirements

1. Force on glider display software for PDA and PC
    a. Recreate image of glider
    b. Recreate orientation of glider
    c. Display g-forces via vector representation on glider in Cartesian space
2. Flight reconstruction software for PDA and PC
    a. Record pitch, roll and acceleration data
    b. Use data to recreate flight path in three dimensional space
    c. Overlay on a map of the area

3.5 Summary

With these requirements we can easily measure the completion of the project. We can compare the final results of the FAADR to the requirements in this section to analyze how well the objectives of this project were completed.

4 System Design

4.1 Introduction

The overall system design of the Flight Attitude and Acceleration Data Recorder is presented in this chapter. The individual input and output of each component as well as the justification of implementation is addressed within the context of the system design block diagram provided in the Overall System Design section. The reason for choosing particular components implemented in this project is discussed in 4.3 Component Selection. The microprocessor software and analysis software design is provided in the Software section.

4.2 Overall System Design

Based on the Problem Statement, we identified several different types of components that are required to meet the goals of this system. These modules are listed here, shown below in Figure 11 as an overall system diagram for the FAADR, and are discussed in greater detail in the following sections.

- Electric Compass
- Accelerometer
- LCD Module
- Microprocessor Unit
- Removable Media Storage Module (SD Card Module)

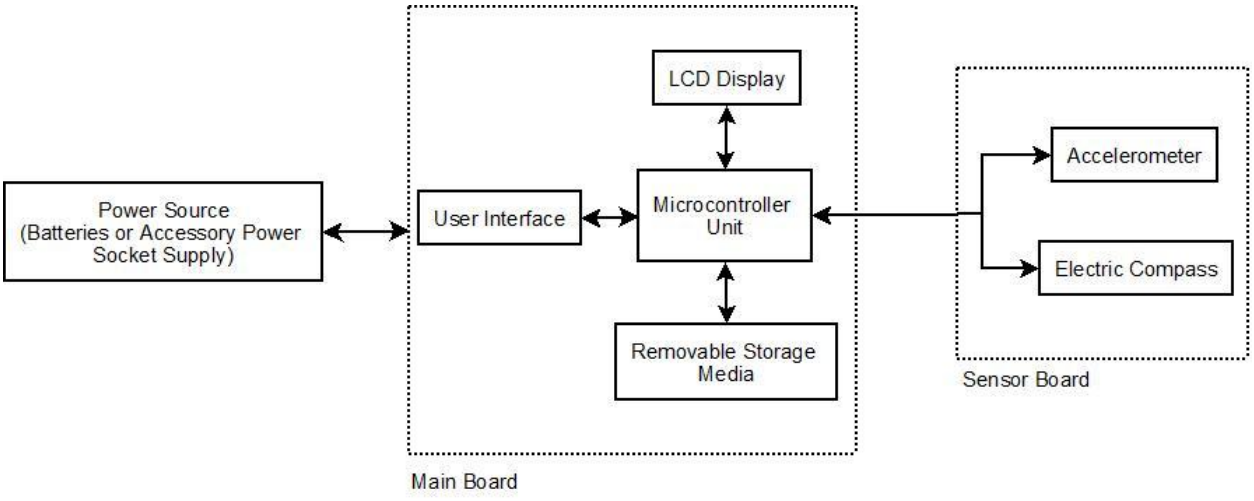


Figure 11: FAADR System Block Diagram

4.2.1 Enclosure

Due to the inherent sensitivity of the electric compass to magnetic fields and the magnetic "noise" provided by the other instruments in the cockpit, the FAADR is contained within two different enclosures. The system's main enclosure is required to fit inside a glider's cockpit. The sensor enclosure houses the accelerometer and the electric compass. It is designed to be placed in the back of the aircraft, away from the noise of the cockpit, and be connected to the main enclosure by an Ethernet (RJ-45) cable. Both enclosures are designed to have Velcro straps attached to the back of the module and their placement inside the cockpit is the pilot's preference. Professor Looft provided us with some insight into his preference and felt that this method of securing the enclosures was adequate.

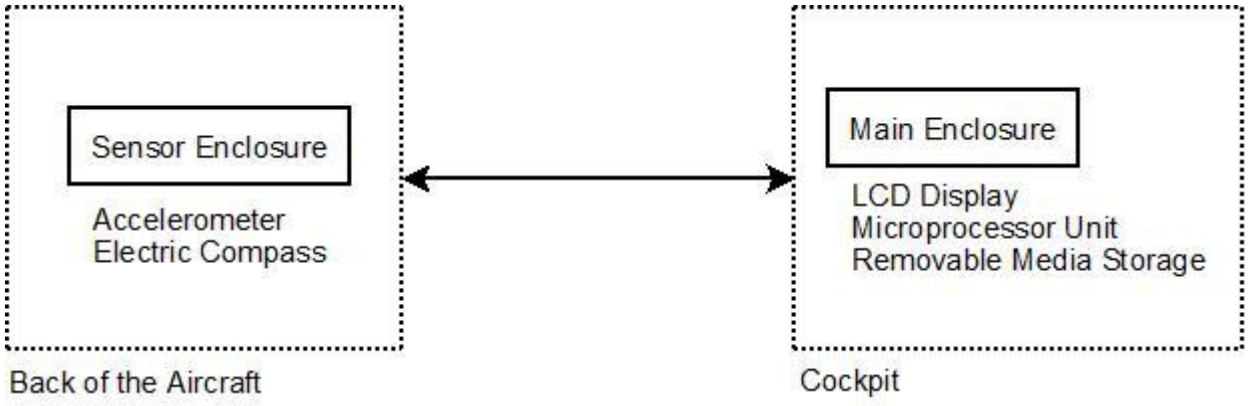


Figure 12: Enclosure Location and Inventory

4.2.2 Electric Compass

The electric compass consists of a tilt and temperature compensated compass and magnetometer that outputs data using a true serial RS-232 protocol. As seen in the Figure 13, the compass requires a 5 Volt power input and communicates through the Ethernet cable to the Microprocessor Unit in the main enclosure. The RS-232 is converted to TTL logic in order for the microprocessor unit to process the information. Including the compass in the system provides us with heading, roll, pitch and temperature data which fulfills part of the system's data acquisition requirements.

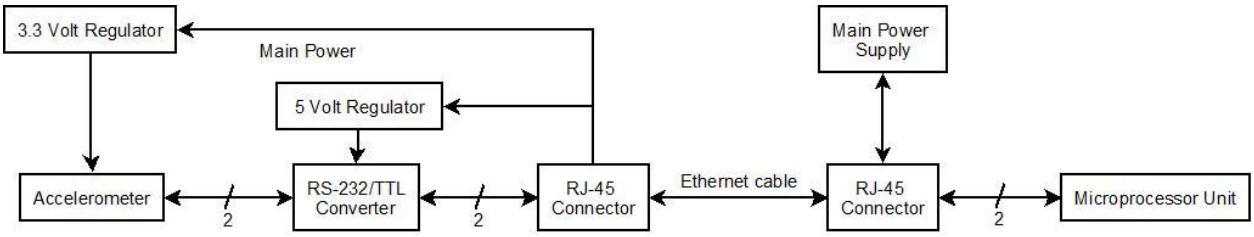


Figure 13: Electric Compass System Block Diagram

4.2.3 Accelerometer

The accelerometer measures acceleration on three axes and provides the system with data that is critical to determining stress on the aircraft. The component must sit on a flat surface in the glider in order to accurately measure individual axis acceleration. For this reason, this component is placed in the sensor enclosure with the electric compass. The accelerometer communicates using I^2C and SPI protocols. The SPI interface (4 wires) was chosen because its speed, simplicity, and because other components also communicate using SPI. For more information about SPI communication, see 2.7 Serial Communication. The accelerometer communicates directly with the Microprocessor Unit through the Ethernet cable between enclosures. The Ethernet cable also carries the 12V supply from the main board which is regulated to a 3.3 Volt supply on the sensor board as shown in Figure 14.

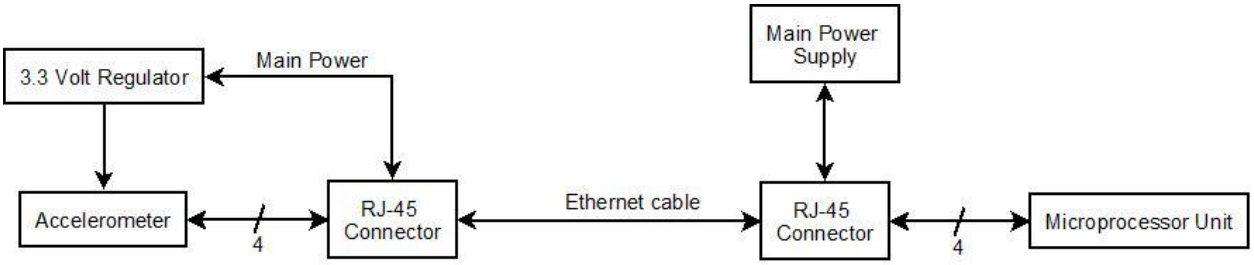


Figure 14: Accelerometer System Block Diagram

4.2.4 LCD Module

The LCD module, whose system is shown in Figure 15, is a graphical liquid crystal display that communicates using a 22-pin parallel interface. It requires a 5 Volt, 3.3 Volt and -16 Volt supply. The -16 Volt supply is internal generated by the LCD module and is to adjust the display's contrast. This contrast supply runs through a variable potentiometer so that the display's contrast can be tuned to the user's preference. The 5 Volt supply is used by the backlight of LCD Display and the 3.3 Volt supply is used to power the LCD display's on-board microcontroller. The Microprocessor Unit communicates directly with this module on the main system board using 11 pins. The LCD module will allow the user to see active readouts of the data acquisition components as well the graphical representation of the systems orientation.

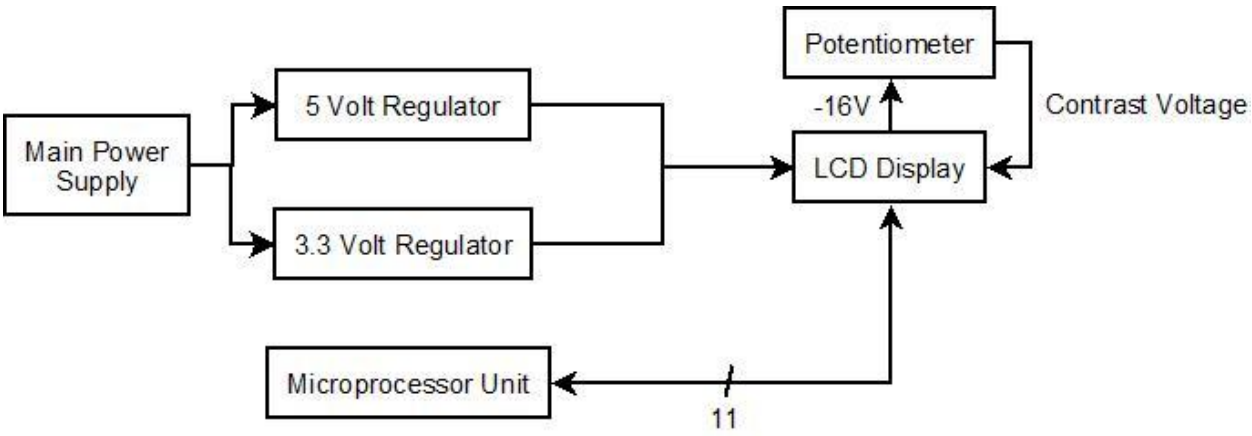


Figure 15: LCD Module System Block Diagram

4.2.5 Removable Media Storage Module

The Removable Media Storage Module, whose system is shown in Figure 16, communicates using UART and SPI protocols over a 12-pin interface and an additional two pins as jumpers to select which communication protocol. It requires a single 5 Volt supply. We use the SPI protocol and connect to the same communication port on the Microprocessor Unit as the accelerometer. This device is not a true SPI device because it introduces an additional two pins to the standard 4-pin interface. This module allows us to store data that we have collected on a non-volatile medium that can be used in the PDAs and PCs so that our analysis software can access it.

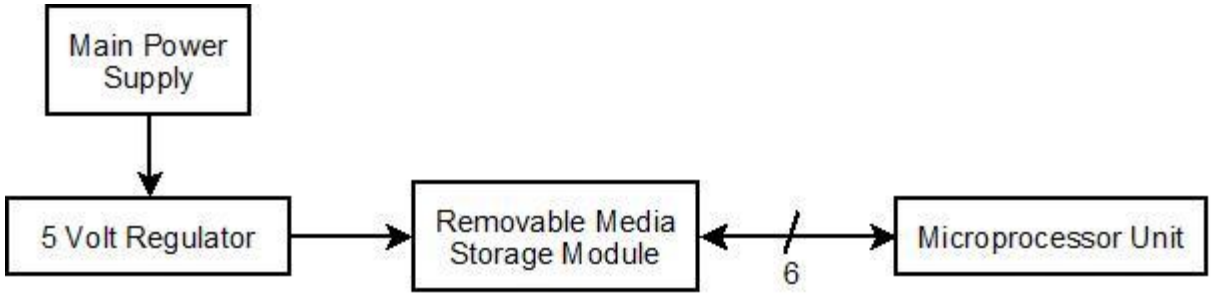


Figure 16: Removable Storage Media Module System Block Diagram

4.2.6 Microprocessor Unit

The Microprocessor Unit is 56-pin board that connects all the components and processes data. As shown in Figure 17, it connects to two SPI devices (Removable Storage Media Module and Accelerometer) on the same communication port, one UART device (Electric Compass) using a separate bus and the LCD module using two IO ports. It requires a single 3.3 Volt supply.

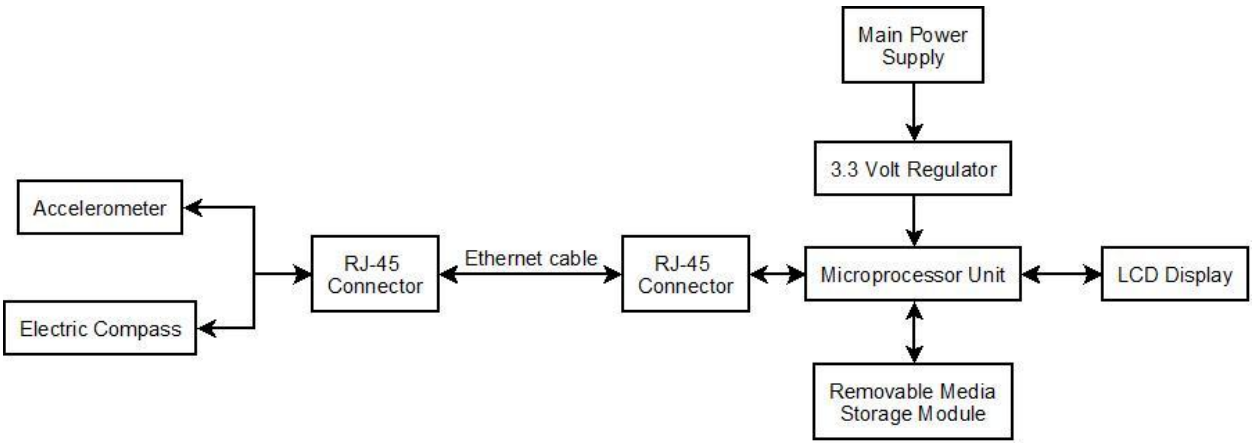


Figure 17: Microprocessor Unit System Block Diagram

4.3 Component Selection

After identifying what components are needed in the system, we assessed what components we had available locally and then analyzed which components were available. For each component we identified a set of minimum requirements that must be fulfilled in order to make the component an option for purchase. We proceeded to perform a cost-benefit analysis on the remaining component options by analyzing the differences between the component, assigning each data point a weight and score, and then calculating a weight average. Several components were chosen because of tools readily available to us through the department.

4.3.1 Electric Compass

Minimum requirements for the system's electric compass were:

- Maximum 2 degree margin of error
- Low power consumption
- Adequate sample rate
- Acceptable price
- Capability of producing X and Y data for pitch and roll computation
- SPI interface is preferable

The table below is the raw component data gathered from their respective datasheets and the price is quoted from Sparkfun.com. We compared 4 different electric compass units the Honeywell HMC6352, PNI Vector V2XE, PNI Micromag2 and PNI Micromag3. All of the devices reported error percentage when determining heading. In order to relate percentage in terms of degrees of error, we applied the follow equation:

$$\text{Degrees of Error} = \frac{\text{Error Percentage}}{100 \times 360}$$

Equation 3: Calculating Degrees of Error from Reported Error Percentage

With this formula the components are found to have the following degrees of error: the Honeywell HMC6352 has 2.52°, the PNI Vector V2XE has 1.98°, and the PNI Micromag2 and Micromag3 both have 12.96°.

Component	**Power**	**Error**	**Interface**	**Sample Rate**	**Price**
HMC6352[10]	1mA @ 3V	0.70%	I²C	20Hz	$59.95
VECTORV2XE	2mA @ 3V	0.55%	SPI	175KHz	$84.95
PNIMICROMAG2	500µA @ 3V	3.60%	SPI	2kHz	$52.95
PNIMICROMAG3	500µA @ 3V	3.60%	SPI	2kHz	$59.95

Table 1: Raw Data for Considered Electric Compass Units

Based on the raw data in Table 1, we rated devices with the scores and produced the data in Table 2. Table 2 is shown graphically in Figure 18. The error percentage for both the PNI Micromag2 and Micromag3 did not meet the minimum requirements and therefore received a minimum score but were still considered. The components were all found to have comparable power consumption. The sample rate for the PNI Vector V2XE was found to exceed our requirements however the price was the most significant. The Honeywell HMC6352 was devalued by a non-SPI interface and the lowest sample rate.

Component	**Power**	**Error**	**Interface**	**Sample Rate**	**Price**	**Score**
HMC6352	8	7	3	4	6	28
VECTORV2XE	8	8	9	10	5	40
PNIMICROMAG2	9	0.1	9	9	6	33.1
PNIMICROMAG3	9	0.1	9	9	5	32.1
Average	8.50	3.80	7.50	8.00	5.50	33.30

Table 2: Raw Scores for Considered Electric Compass Units

Since error is a requirement and low power consumption is goal, both of these data points received a heavy weight. Sample rate is also a minimum requirement and therefore weighted heavily. Interface and price were both important but not necessary.

After scoring the electric compass units we applied the weights as shown in the first row of Table 3 based on the criteria shown in Table 4 and came up with the weighted scores shown in Table 3 and Figure 19. The results conclude that the PNI Vector V2XE, shown in Figure 20, was the best choice despite having the highest price. This unit was also the only compass to fully meet the minimum requirements for the system's electric compass.

Weight	0.7	1	0.6	0.8	0.5	
Component	**Power**	**Error**	**Interface**	**Sample Rate**	**Price**	**Score**
HMC6352	5.6	7	1.8	3.2	3	20.6
VECTORV2XE	5.6	8	5.4	8	2.5	29.5
PNIMICROMAG2	6.3	0.1	5.4	7.2	3	22
PNIMICROMAG3	6.3	0.1	5.4	7.2	2.5	21.5
Average	5.95	3.80	4.50	6.40	2.75	23.40

Table 3: Weighted Score for Considered Electric Compass Units

	0-2	**3-5**	**6-8**	**8-10**
Power (mW)	> 30	20 - 30	10 - 20	< 10
Error (%)	> 1.5	1 - 1.5	0.5 - 1	0 - 0.5
Interface	None	I^2C	SPI	
Sample Rate (Hz)	< 2	2 - 10	10 - 100	> 100
Price ($)	> 100	75 - 100	50 - 75	< 50

Table 4: Scoring Chart for Electric Compass Units

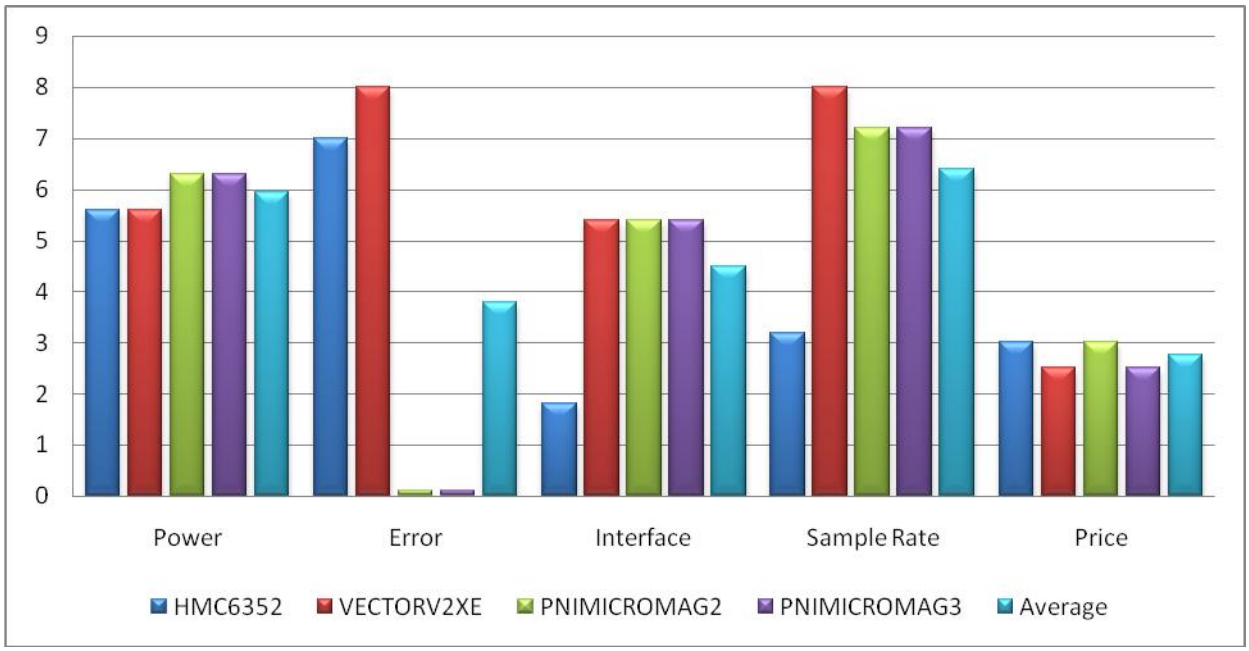


Figure 18: Chart of Electric Compass Unit Scores

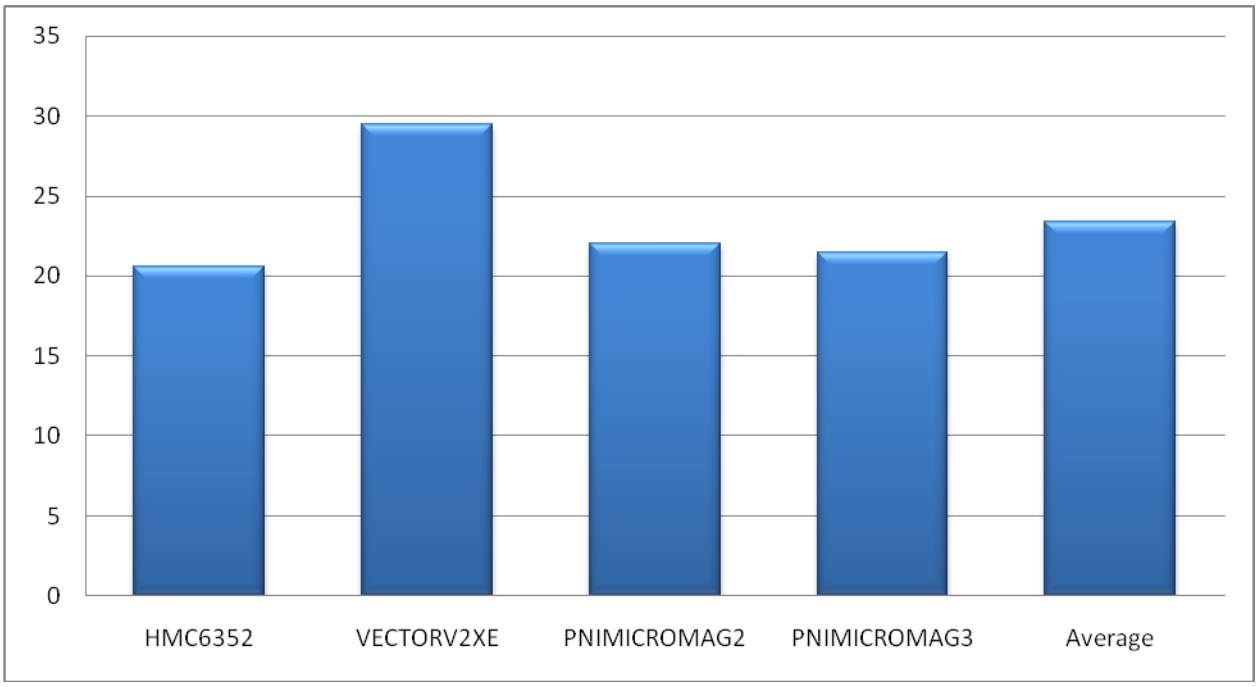


Figure 19: Chart of Weighted Scores of Electric Compass Units

Figure 20: The V2Xe 2 Axis Digital Compass (http://www.pnicorp.com)

4.3.2 Accelerometer

The minimum requirements for system's accelerometer are:

- ±5 G range
- ±0.1 G error margin
- Adequate sample rate
- Low power consumption
- Acceptable price

The complexity of the component's system in regards to axes measured was taken into account. A two-axis accelerometer would require at least two units to be capable of making measurements of all three

axes. This would also involve precise orthogonal mounting of the two units to ensure the accuracy of the data. We compared three different accelerometers:

- Analog Devices ADXL320
- Analog Devices ADIS16006
- VTI Technologies SCA3004-E04.

The number of axes measured factor is reflected in the Table 5 and shown in terms of adjusted price and complexity. The accuracy in terms of Gs can be determined by multiplying the range of measurement by the accuracy. This shows that the ADXL320 has a margin of error of 0.01 G, the ADIS16006 has a 0.25 G margin of error and the SCA3000-E04 has a 0.03 G margin of error.

Name	**Range**	**Accuracy**	**Base Price**	**Complex.**	**Adj. Price**	**Sample Rate**	**Current**	**VDC**	**Power**
ADXL320	±5G	±0.2%	$3.75	2	$7.50	2.5kHz	0.5mA	3V	1.5mW
ADIS16006	±5G	±0.5%	$17.75	2	$35.50	2.3kHz	1.2mA	3.3V	5mW
SCA3000E04	±6G	±0.5%	$27.00	3	$27.00	100Hz	0.12mA	2.5V	0.3mW

Table 5: Raw Data for Considered Accelerometers

Component	**Range**	**Accuracy**	**Adj. Price**	**Sample Rate**	**Power (mW)**	**Complexity**	**Score**
ADXL320	8	9	9	9	10	6	51
ADIS16006	8	8	3	9	10	6	44
SCA3000-E04	9	8	5	8	9	9	48
Average	8.33	8.33	5.67	8.67	9.67	7.00	47.67

Table 6: Raw Score for Considered Accelerometers

Table 6 and Figure 21 show that highest unranked score belongs to the ADXL320 due to its high accuracy and low power consumption. However both the ADXL320 and ADIS16006 are two-axis accelerometers and are penalized for their increased complexity. The SCA3000-E04 is the only three axis accelerometer and has a slightly higher G range than required but is significantly more expensive than the other units.

Table 7 shows that range and accuracy are rated highly because they are both minimum requirements for the unit. Complexity is also weighted heavily because of the costs in terms of additional work and the area consumed by the unit. Power is a goal and considered important. Sample rate is important but all of the units met the minimum rate.

Weight	1	0.9	0.5	0.6	0.6	1	
Name	**Range**	**Accuracy**	**Adj. Price**	**Sample Rate**	**Power**	**Complexity**	**Weighted Score**
ADXL320	8	8.1	4.5	5.4	6	6	38
ADIS16006	8	7.2	1.5	5.4	6	6	34.1
SCA3000-E04	9	7.2	2.5	4.8	5.4	10	38.9
Average	8.33	7.50	2.83	5.20	5.80	7.33	37.00

Table 7: Weighted Scores of Considered Accelerometers

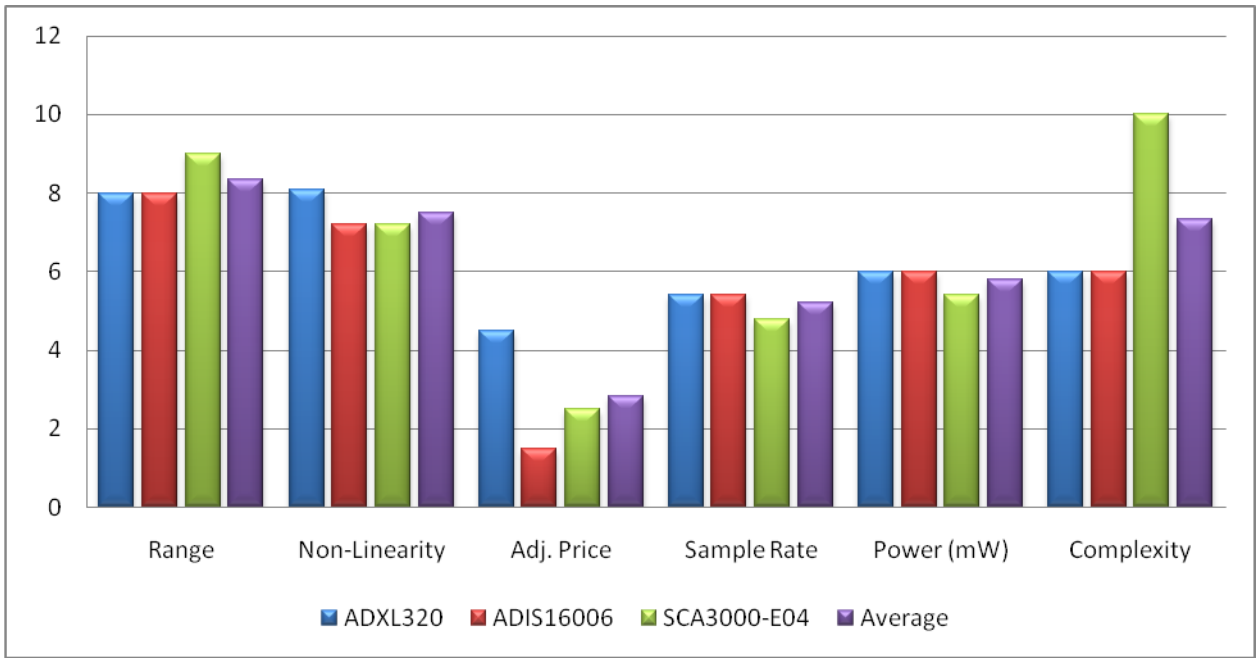


Figure 21: Chart of Considered Accelerometers Unit Scores

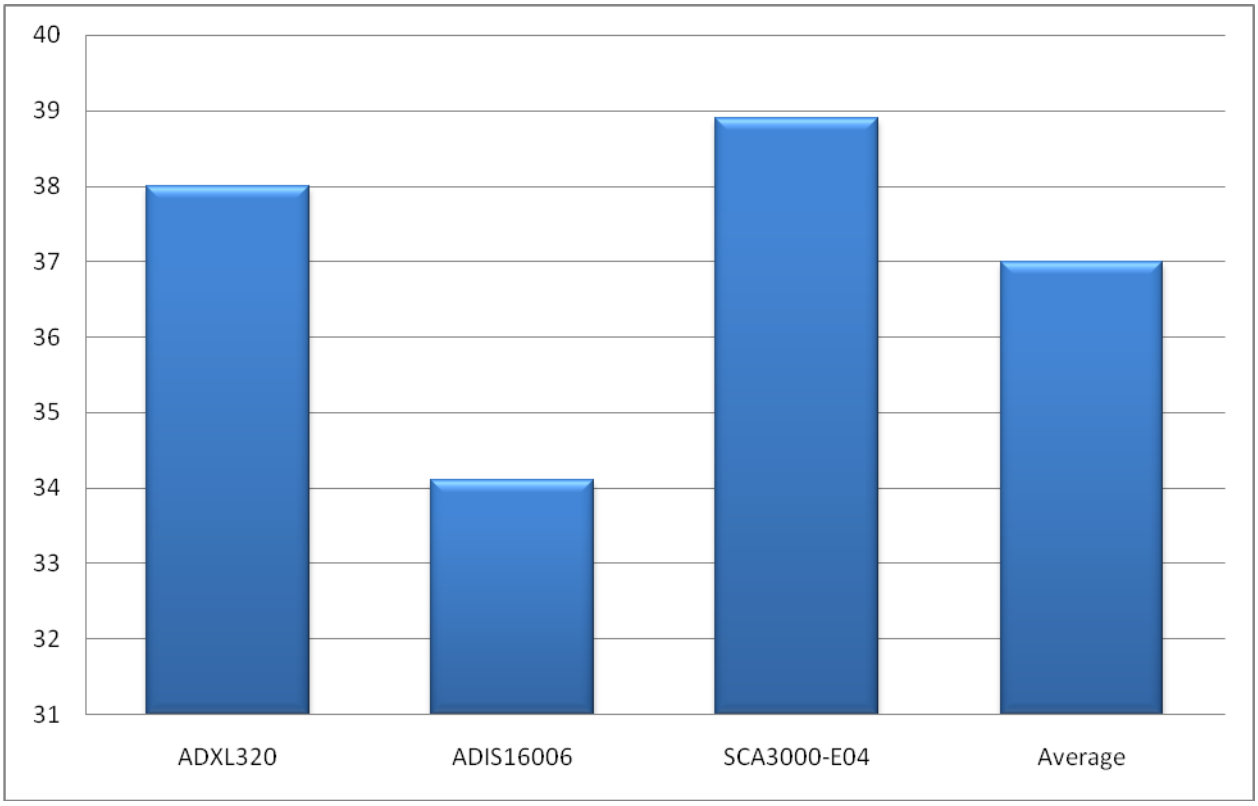


Figure 22: Chart of Considered Accelerometers Total Scores

Figure 22 clearly shows that the highest scoring accelerometer, after applying the weights, belongs to VTI Technologies' SCA3000-E04, shown in Figure 23, despite the price. This was extremely desirable because of its ability measure all three axes in one unit and the lowest power consumption of the units considered.

Figure 23: SCA3000-E04 3-Axis Accelerometer

4.3.3 LCD Module

The LCD module that we choose for this project was CrystalFontz CFAG128128A-STI-TZ graphical liquid crystal display. This manufacturer was recommended to us by Professor Looft because of the success his student have had with these displays in previous projects. When we began to determine the requirements for this module we realized that most of these should be determined by the customer. Therefore, Professor Looft is primarily responsible for choosing this unit.

Figure 24: CFAG128128A-STI-TZ from CrystalFontz America Inc.

We compiled a set of displays by this manufacturer that we found to be acceptable based on size, resolution and power consumption. All of the displays had a similar hardware and software interface. We presented this set to Professor Looft and asked him to determine the display that he preferred out of that set.

4.3.4 Microprocessor Unit

The Microprocessor Unit that we choose was the MSP430F1491 on SoftBaugh B1491 breakout board. We choose the MSP430 because of the design tools available to us already on campus such as the IAR Embedded Workbench software development package. The system's microcontroller was required to have at least two SPI/UART ports and a substantial amount of memory. The MSP430F1491 was the only chip that satisfied these requirements with 60KB of flash memory and 2 SPI/UART ports.

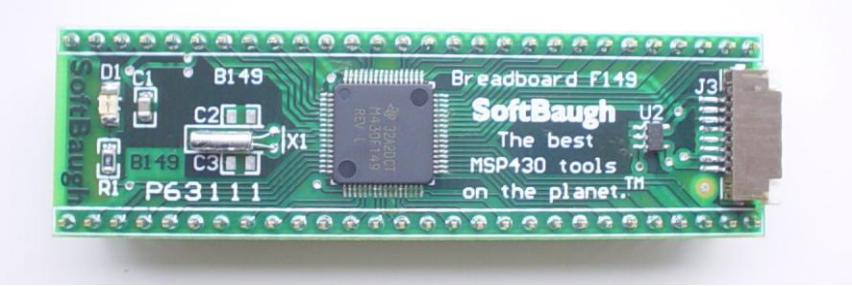


Figure 25: B1491 Breakout Board for the MSP430F1491

After we decided what chip to purchase, we found the manufacturer SoftBaugh that made breakout boards for the MSP430 chips and had a USB programmer available. IAR Embedded Workbench had recently released drivers for Softbaugh's programmers so we decided to purchase the B1491 which had the external oscillator and JTAG connection already configured on the board itself.

4.3.5 Removable Media Storage Module

The Removable Media Storage Module that we choose was the SparkFun.com Breakout Board for the DOSonCHIP FAT16/FAT32. We had decided that the storage medium must be an SD Card so that it could be easily used by a PC and a PDA. Sparkfun.com offered three different types of SD Card breakout boards. The DOSonCHIP offered the CB1710 chip which processed DOS-like commands using a SPI protocol and boasted low power consumption.

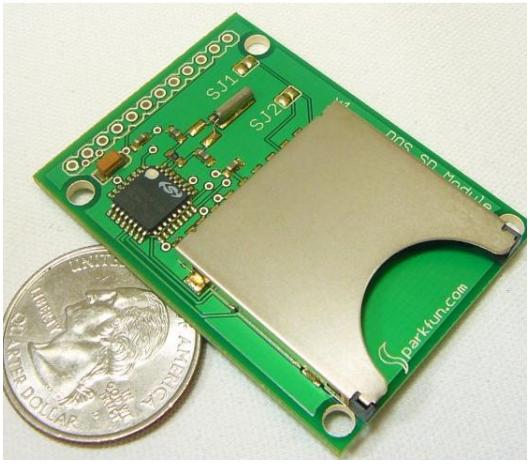


Figure 26: Breakout Board for the DOSonCHIP FAT16/FAT32 Module

4.4 Summary

In this section we described each component required to fulfill the requirements listed in Section 3 Problem Statement. Each individual component's system is defined and its position within the overall system shown. This chapter also shows each component that was purchased for the FAADR and why that component was chosen.

5 Results

5.1 Introduction

This section describes hardware and software implementation for each module which were combined to create the data recorder. After presenting how the modules were implemented, the results of the individual components and the data recorder will be given.

5.2 Hardware Implementation

Designing a system that worked completely as intended required that each component described in the system design section must work exactly as intended and interface correctly with the system as a whole. The components required to create the FAADR needed to be implemented and tested individually to ensure that they functioned as standalone units. Each component then needs to be integrated to ensure that can cooperate with the rest of the system. The components that were implemented and tested follow:

- MSP430F1491 Microcontroller
- Crystalfontz Graphic LCD
- DOSonCHIP-SD Module
- Angular Rate Sensor MLX90609
- SCA3000-E04 3-AXIS ACCELEROMETER

5.2.1 MSP430F1491 Microcontroller

We first needed to have a functioning microcontroller so that we could successfully interface with the other components. Testing of the microcontroller required that we ensure operation of the following items:

- Softbaugh USBP Programmer/Debugger
- Software Implementation
- Logical Output
- Logical Input

While the testing of these items seemed at first to be a large task, it was simplified by the fact that all of these tests could be performed with a small program uploaded to the MSP430. We had chosen to write our first program for the module using IAR Embedded Workbench since it could interface directly with the processor via the Softbaugh USBP Programmer. Using IAR, we wrote a program in C++ which would enable a pin which was connected to an LED (internally on the module) and disable the pin via an infinite loop. The C++ code for this test algorithm can be found in Appendix A. Since the pin was enabled and

disabled repeatedly, we would be able to see the LED flashing which would validate that our processor board could be programmed and output data.

Validating that the MSP430 could accept inputs was equally as simple. To ensure that inputs functioned adequately, we wrote a small program which utilized the same LED pin described previously. The code would simply enable the LED when logic high was generated on the pin and disable the LED when a logic low was applied to the pin. The C++ code for this test algorithm can be found in Appendix A. Once the code was uploaded to the MSP430 we could test the input functionality of the board simply by either applying a logic high (3.3 Volts) or a logic low (0 Volts) and ensuring that the LED responded accordingly.

Since both the input and output algorithms generated the correct LED outputs we could be sure that the MSP430 could:

- Successfully be programmed via the Softbaugh programmer
- The software would behave as would be expected
- The processor correctly handled inputs to its pins
- The processor correctly handled output to its pins

Knowing that our MSP430 board could adequately perform these functions we could move on to testing our other modules with this board. Since the MSP430 has been validated, we could be reasonable sure other modules are the source of any errors encountered during testing.

5.2.2 Crystalfontz Graphic LCD

Testing the graphic LCD from a strictly hardware standpoint would require the ability to power the device to an ON state and then display an intended character on the screen. Powering the device to a state where we could be sure the screen was ON required that we connect pins VDD and LED+ to a 5 Volt supply while grounding the required pins. Once these pins were connected the LCD backlight was illuminated and the screen output could be read.

Displaying a character on the screen of the LCD would be much more challenging since we needed to interface our MSP430 controller to the T6963C controller that resided on the LCD board. The interface we needed to use, in order to communicate with the display, was parallel which meant that we needed to put a character on the bus, enable the display and then signal the read pin to read our characters into memory. The schematic for this interface between the MSP430 and the T6963C can be viewed in Figure 27.

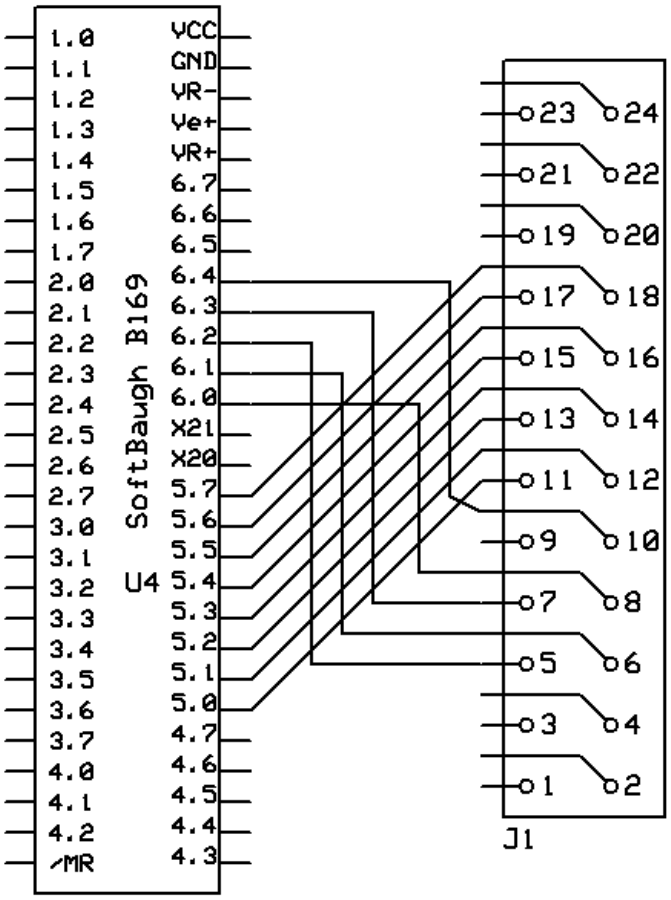


Figure 27: Schematic Diagram of MSP430 to LCD Display Interface

Figure 27 shows the data bus, which holds the character for the LCD display, comprises pins 11 through 18, whereas, the control logic for loading the character into the LCD board's memory comprises pins 5 through 10. The control logic for the LCD module will be discussed in greater detail in the software implementation section.

Using the connections shown in Figure 27, we wrote code that can be found in Appendix A that simply writes a character to the screen of the display. At this point, the team reached the first problem with testing of a component since the character was not being displayed on the screen. Analyzing the command and data signals that were being sent to the LCD revealed that the correct data was being sent to the T6963.

At this point we logged onto the Crystalfontz forums to see if any other users of this particular display were having similar problems. Luckily, we found a solution to the problem quickly since many other people seemed to be having similar problems. The problem was found in the contrast of the screen. In order to control the contrast of the screen, a 10K Ohm potentiometer needed to be placed between pins 4 and 20 (LCD power supply driver and -16 Volt output) of the LCD display. By tuning the potentiometer so that the contrast was at a viewable level, we could now view the character which was written by the previous command.

This test confirmed that the LCD display was working and that we could successfully send signals to the display driver without having to worry that the display was functioning incorrectly. Since we could write to the screen, this would give us a perfect tool in order to test and debug other components.

5.2.3 DOSonCHIP-SD Module

The DOSonCHIP module was rather simple to test since all commands that could be sent to the chip mirror DOS commands for opening, closing and writing to files. To ensure that this module was functioning correctly, we needed to be able to accomplish the following:

- Write code to initialize the SPI interface of the MSP430
- Validate signals sent from the MSP430
- Connect the SPI bus of the MSP430 correctly to the SD module
- Send a command to the SD module
- Verify that the command was received by the module

To carry out this test, we first needed to initialize our MSP430's SPI interface which was a time consuming task since there were many different registers that needed to be configured in order to ensure correct operation. The initialization of the SPI interface is described in Section 5.3.

Once properly initializing the SPI port, we could connect the SD card module to the MSP430. Figure 28 shows how the MSP430 is connected to the SD card module.

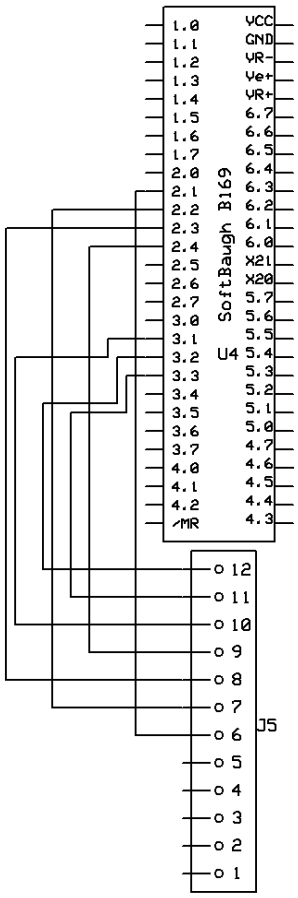


Figure 28: Schematic diagram of DOSonChip and MSP430 Microcontroller interconnection

Once the SPI bus was initialized we needed to send a series of ASCII characters which would be the initial command sent to the SD module. The command sent was “ow” which stands for open write. To send the characters to the module, we needed to simply put the command into the transfer buffer of the MSP430 one byte at a time. After the “ow” command and the path of the file to open was sent, we put a final carriage return character into the transfer buffer to signal the module that the command was complete.

If the module received the command and could successfully open the file to write, the module would send back a file handle, (typically 1, if no other files are open) which would allow us to verify that the module was functioning. At first we did not receive any signals back from the module. Since documentation for this particular module was woefully inadequate, we needed to email support with our configuration and problem.

After a series of emails clarifying the situation, we were informed that we needed to check the ‘busy’ and the ‘Dir’ pins of the module so that we knew when to check for input and when to send output, respectively. Although this information was no doubt helpful for both speed and stability of the software, it did not solve the problem of the SD module not sending back a file handle.

After another series of emails to support, we were asked in what mode the module was set. We did not set a mode for the module because there were no indications that this needed to be done in the documentation. We were informed that there was a mode contact on the top of the board which could be grounded or pulled high to specify if we were going to use the SPI or UART mode of the device. After setting the mode of the device to SPI we successfully retrieved a file handle from the SD module.

This test showed that we had verified a large portion of the functionality of the SD module and ensured that the module was in working condition. We had successfully demonstrated that we could initialize an SPI port correctly, send characters, and receive input via the SPI receive buffer. We had now verified that all outputs from our system (LCD and SD) were functioning. Now we needed to verify that the inputs to our system (compass and accelerometer) were also functioning.

5.2.4 SCA3000-E04 3-Axis Accelerometer

The accelerometer, shown in Figure 23, which would read the forces which were acting on the glider, needed to be tested to make sure that it was successfully operating. This module also used an SPI interface so it would be rather simple to reuse the SPI code from the SD Card for the accelerometer. One difference between the SPI initialization between the accelerometer and the SD module is clock polarity that needed to be used for the accelerometer was the inverse of the SD Card polarity. Changing of the clock polarity will be discussed in Section 5.3 Software Implementation. The connection of our MSP430 to our accelerometer via an Ethernet port is shown in Figure 29 and Figure 30.

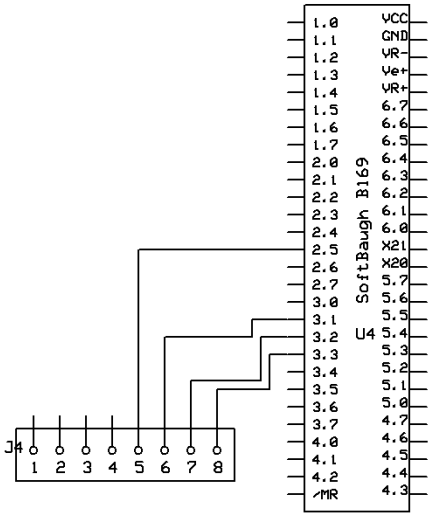


Figure 29: MSP430 connections to Ethernet port which correspond to accelerometer connections

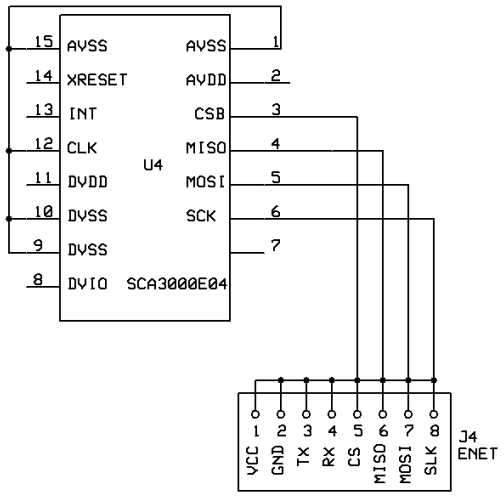


Figure 30: Accelerometer connections to Ethernet port which correspond to MSP430 connections

The steps that needed to be taken to verify the accelerometer functionality included:

- Successfully send SPI message to the accelerometer requesting that the X-axis acceleration be sent back
- Retrieve the X-axis acceleration frame from the accelerometer
- Parse the frame so that the axis data can be retrieved
- Convert the data into a valid integer to check the result

Sending an SPI message to the accelerometer was rather simple since we already had practice sending SPI messages to the SD module. In order to request information from the accelerometer, we needed to send the address of the information which we wanted to have sent back. Since the x-axis value was stored at location 05h, we sent 05h to the accelerometer and retrieved back the value of the x-axis acceleration.

After checking the receive buffer for the acceleration value, we needed to convert the number from the form it was given in, to an integer value. Since the value form was unclear and not well documented we engaged the company's (VTI) support and received a spreadsheet and an equation on how to convert the number. This equation was implemented in code as can be seen in Appendix A: Source Code and allowed us to view seemingly correct acceleration values being sent back from the accelerometer.

Being able to request, retrieve and convert acceleration data from the module proved that the accelerometer was functioning correctly and could provide us with the data that we would need to display for the end user.

5.2.4 Angular Rate Sensor MLX90609

The component that would sample the angle of the unit allowing it to create an artificial horizon went through the most change in this project. We attempted to implement three separate components throughout the course of the project which include:

- V2Xe 2 Axis Digital Compass
- EZ-Compass-3
- Angular Rate Sensor MLX90609

Shown below in Figure 31, the first component that was implemented in an attempt to sample pitch and roll vectors was the V2Xe 2 axis digital compass. Testing this compass was quite similar to the testing that the accelerometer unit went through. As with the accelerometer, we had already setup the SPI port on our microcontroller so we could easily begin to communicate with the compass and read in heading. After several readings of the 2 axis compass, we were confident that we could retrieve accurate pitch and roll from this unit.

Figure 31: The V2Xe 2 Axis Digital Compass (http://www.pnicorp.com)

The compass however was not tilt compensated, which means that when the compass is tilted, both the pitch and the roll values become in correct. The lack of tilt compensation was apparent when the compass was tilted to its side. When on its side, the unit would return widely varying values for pitch and roll, so the team abandoned the idea of using this compass and asked Professor Looft if he could suggest any alternatives.

Figure 32: AOS EZ-Compass 3 (www.aositilt.com)

Professor Looft happened to have an EZ-Compass-3 (shown in Figure 32) which we could use to gather pitch and roll data since it was a 2 axis tilt compensated compass. This was a large improvement over the V2Xe compass since it would not skew values when it was tilted on the third axis. This compass would need to be tested to make sure that we could properly interface with the unit.

From reading through the product's datasheet, it was found that the compass would need to communicate with our MSP430 via a UART port. The MSP430 still had one open UART port that could be used for the compass. Testing the compass with the UART port was rather simple since the initialization was quite similar to SPI initialization. The code for this initialization can be found in Appendix A. The major differences between using UART and SPI for communication were that we could simply transmit and receive to the compass without having to enable a chip select on the compass. The UART communication proved that the compass would be able to return the correct values for pitch and roll which the project required. However, in the latter portion of the project schedule, the compass began sending no data when it was queried.

The compass was tested extensively after it began returning no data by both communicating with our MSP430 and via a HyperTerminal connection with a PC. According to the support for the EZ-compass-3 the HyperTerminal connection to the compass should immediately show if the compass is working because every time that the compass is powered on it will send a header at 19200 baud. We found that the header was no longer being sent from the compass which led to the conclusion that somehow the compass was damaged over the course of its testing and we would need to seek an alternative component. We could not simply buy another compass because 2 axis tilt compensated compasses are too expensive to fit the budget given to this project.

The final component that was tested for use with our project was the Melexis MLX90609 Angular Rate Sensor MLX90609 which is shown in Figure 33. This module was far different from the modules we

were previously using since it was neither a compass and nor strictly a serial device. The rate sensor simply detected the rate of angular change that it experienced along one axis and reported that change via a voltage on one of it pins. This would be simple to interface with and test since our MSP430 had several analog to digital converters (ADC). To test this new component, we needed to connect the rate sensor to an A/D converter and then sample the data received.

Figure 33: The angular rate sensor MLX90609 (http://robosavvy.com)

In order to accurately retrieve the orientation of the aircraft, the rate of change data transmitted by the sensor needs to be manipulated. To do this we needed to offset the value that was being returned by the unit, divide the value by a constant to ensure degrees per second and then integrate the value over time to obtain change in angle which are all described in detail in Section 5.3 Software Implementation.

After putting a large amount of effort into testing and verifying the units which could have been acceptable for our project, we found that the rate sensor was the only viable component that could be used to gather accurate data for both pitch and roll since it was not affected by tilting on the third axis. After finding that the rate sensor would be used for the task we needed to write the software which would allow the MSP430 to interrupt and sample the sensor enough so that large changes in angle would not be missed. The description of the software implementation can be found in Section 5.3 Software Implementation.

5.3 Software Implementation

This section describes the implementation and detailed design of the source code of the microcontroller and analysis software. The following sections describe each of the components, the source code associated with that component and the program flow of that source code.

5.3.1 Accelerometer

The accelerometer constructor performs the initialization of the SPI port that communicates with the accelerometer. This code, which is shown in Figure 34, is extremely important for the interface between the microcontroller and the accelerometer to work properly.

```
Accelerometer::Accelerometer() {
    P3SEL |= BIT0 + BIT1 + BIT2 + BIT3; // Setup P3 for SPI mode
    U0CTL = CHAR + SYNC + MM + SWRST; // 8-bit, SPI, Master
    U0TCTL = CKPH + SSEL0 + STC; // Polarity, SMCLK, 3-wire
    U0BR0 = 0x03; // SPICLK = ACLK/3 = 32K/9600 = 3
    U0BR1 = 0x00;
    U0MCTL = 0x00;
    ME1 = USPIE0; // Module enable
    U0CTL &= ~SWRST; // SPI enable
    IE1 |= URXIE0 + UTXIE0; // RX and TX interrupt enable
}
```

Figure 34: Accelerometer Initialization Code

The first line involves activating the three pins that correspond to the necessary SPI signals needed for communication. The following line sets up the SPI controller to operate with 8-bit characters, as the master device and to use the software reset. The U0TCTL register is setup so the output signal will rest at a high logic level, use the external oscillator as the source for its clock divider, and operate in three wire mode. U0BR0 sets the clock divider to divide its source signal by 3, which gives the SPI port a baud rate of about 9600. ME1 enables the entire SPI module and by clearing the software reset bit in the U0CTL register, the SPI port is effectively powered on. The final line in this initialization enables an interrupt to occur when the transmit buffer is cleared and the receive buffer is full.

The accelerometer primary function is the read function. The read function takes in the data structure "accelerometerData" as described in the Accelerometer Header file and fills in the X, Y, and Z acceleration data fields by calling the read register function. The function then calls itoa and xtoa to translate the number that was received from accelerometer into a character array that can be displayed on the LCD and written to the SD card. This flow is shown in Figure 35.

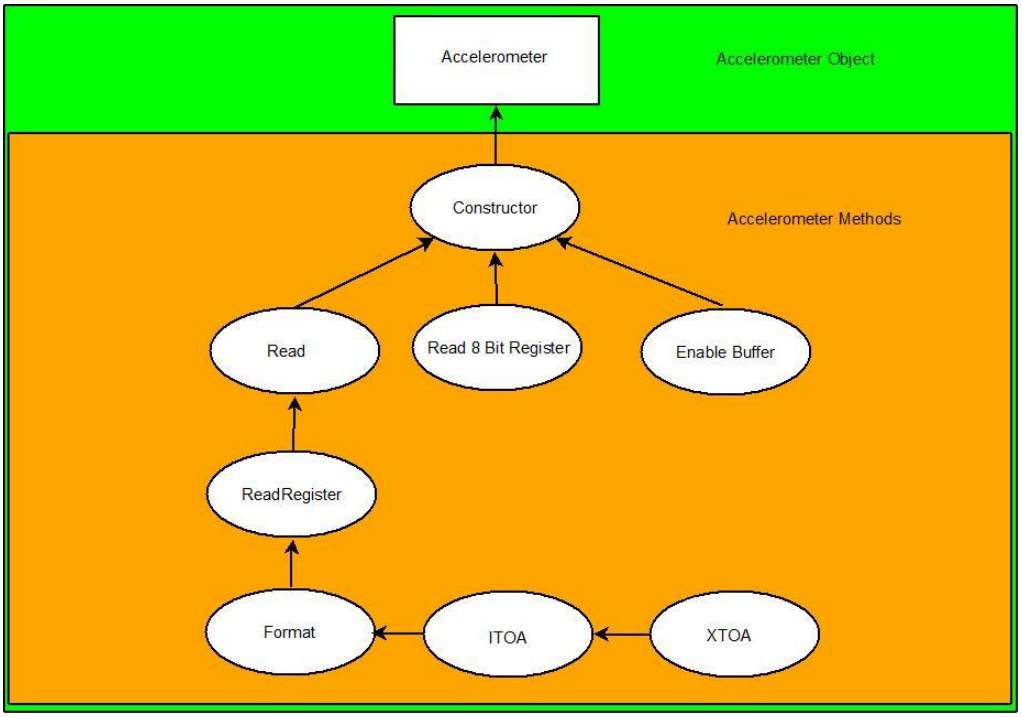


Figure 35: Accelerometer Software Diagram

The read register sends a command to read a register on the accelerometer that is specific to the desired axis. The accelerometer contains several registers that contain data and status information and each register is assigned a byte address that is documented in the datasheet. This read register function is taken directly from the VTI Technologies support website's sample code. This code is used to translate the 16-bit data packet whose structure is shown in Figure 36 that is stored in two 8-bit registers as the most significant and least significant byte of the data.

Byte	MSB byte								LSB byte					
Bit number Acceleration [mg]	B7 Sign	B6 4096	B5 2048	B4 1024	B3 512	B2 256	B1 128	B0 64	B7 32	B6 16	B5 8	B4 4	B3 2	B2:B0 xxx
SCA3000-E04 [X_LSB…Z_MSB]	s	d11	d10	d9	d8	d7	d6	d5	d4	d3	d2	d1	d0	xxx
SCA3000-E04 Ring buffer in 11-bit mode [BUF_DATA]	s	d9	d8	d7	d6	d5	d4	d3	d2	d1	d0	x	x	xxx
SCA3000-E04 Ring buffer in 8-bit mode [BUF_DATA]	s	d6	d5	d4	d3	d2	d1	d0	x	x	x	x	x	xxx

s = sign bit
x = not used bit

Figure 36: Bit Structure of the SCA3000-E04 16-bit Registers

Figure 37 shows the signal generated on the SPI by this function. As you the MSB register contents are requested first and then contents of the LSB. In order to read one byte of data, two writes occur. The register code is written by the master and then a dummy write of a logic low. This allows the clock to be

generated so that the master can receive the slave's output. The SCA3000-E04 register codes are organized so that the value of the code for the LSB is exactly one less than the value for the code for the MSB which makes reading both very simple. For more information on SPI communication, refer to Section 2.7 Serial Communication.

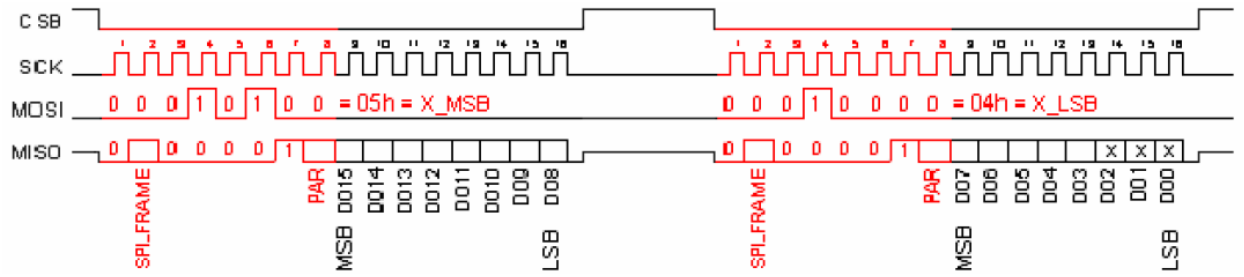


Figure 37: 16-bit Register Read from the SCA3000-E04

The one very important modification we performed to the original read register function is the wait that occurs between reading the MSB and LSB register. We found that the URXIFG0 bit on the MSP430F1491 which is supposed to be an interrupt generated when an entire character is received on the SPI bus activated before the character was fully received which caused us to receive incorrect data. Despite a long dialogue that we generated with VTI support to correct this problem it turned out to be an issue for the microcontroller that was unofficially fairly well known but officially very poorly documented. The solution that we used is very simple volatile variable iterator that causes the microcontroller to wait a small period of time before accessing the SPI receive buffer. While we consider this to be a very inelegant way of bypassing this issue, it does work and we could not find a better way to do it.

5.3.2 Rate Sensor

The rate sensor source code deals with the two analog rate sensors that are located on the sensor board. These two sensors are output to the MSP430F1491's analog-to-digital (A/D) converter. The digital readings are then manipulated and integrated in order to transform rate into position. For accuracy, an interrupt is used that is generated by the MSP430F1491's internal timer, Timer A. The frequency at which the timer interrupts is the sampling frequency of the rate sensor analog signal. The initialization code for the A/D converter and the interrupt is located in Rate Sensor Source and shown in Figure 38.

```
rateSensor::rateSensor() {
    P6SEL = 0x0F; // Enable A/D channel inputs
    ADC12CTL0 = ADC12ON+MSC+SHT0_2; // Turn on ADC12, set sampling time
    ADC12CTL1 = SHP+CONSEQ_1; // Use sampling timer, single sequence
    ADC12MCTL0 = INCH_0; // ref+=AVcc, channel = A0
    ADC12MCTL1 = INCH_1; // ref+=AVcc, channel = A1
    ADC12CTL0 |= ENC; // Enable conversions
    ratePitch = 0;
    rickRoll = 0;
    rateOffset = 0;
    rateCounter = 250;
    rateTotal=0;
    ADC12CTL0 |= ADC12SC;
    P1DIR |= 0x01; // P1.0 output
    CCTL0 = CCIE; // CCR0 interrupt enabled
    CCR0 = 2000;
    TACTL = TASSEL_2 + MC_2; // SMCLK, contmode
    rateOffset = checkPitch();
    _BIS_SR(LPM0_bits + GIE); // Enter LPM0 w/ interrupt

}
```

Figure 38: Rate Sensor Initialization Code

The first six lines of this code are taken directly from sample code provided by Texas Instruments for the MSP430 microcontroller family. They effectively enable the pins, power on the A/D converter, set an internal sampling time, enable two analog channels and enable the entire module. The variables rickRoll, ratePitch, rateCount, and rateOffset are global variables that interrupt uses to integrate the signals.

Setting the CCIE bit of CCTL0 enables Timer A and CCR0 is the number of clock ticks till the interrupt is generated. The internal clock frequency of our MSP430F1491 is 1 MHz which means that an interrupt will be generated at a rate 5kHz. We tried several different rates and found that 5kHz produces a low amount of error but significantly lowers the performance of the microcontrollers other functions.

The remaining lines initialize the interrupts working variables and set the timer's clock divider source signal. The last line of this code tells the MSP430 to effectively shut off the CPU when the timer interrupt is generated. Users of the A/D converters on MSP430 have found that by shutting the CPU down before sampling the analog signal it reduces the noise and results in a more accurate digital value. LPM0 is a particular sleep mode available on the MSP430 that still allows interrupts to be generated and processed. The last operation performed by the interrupt is to reactivate the CPU by clearing the LPM0 bit.

The software diagram in Figure 39 shows how the Rate Sensor Source operates. After the constructor initializes the A/D converter and timer, the rate sensor object that is created operates on its own. The get functions are used by the main class to acquire the current pitch and roll data. All of the processing in this class is done inside the interrupt.

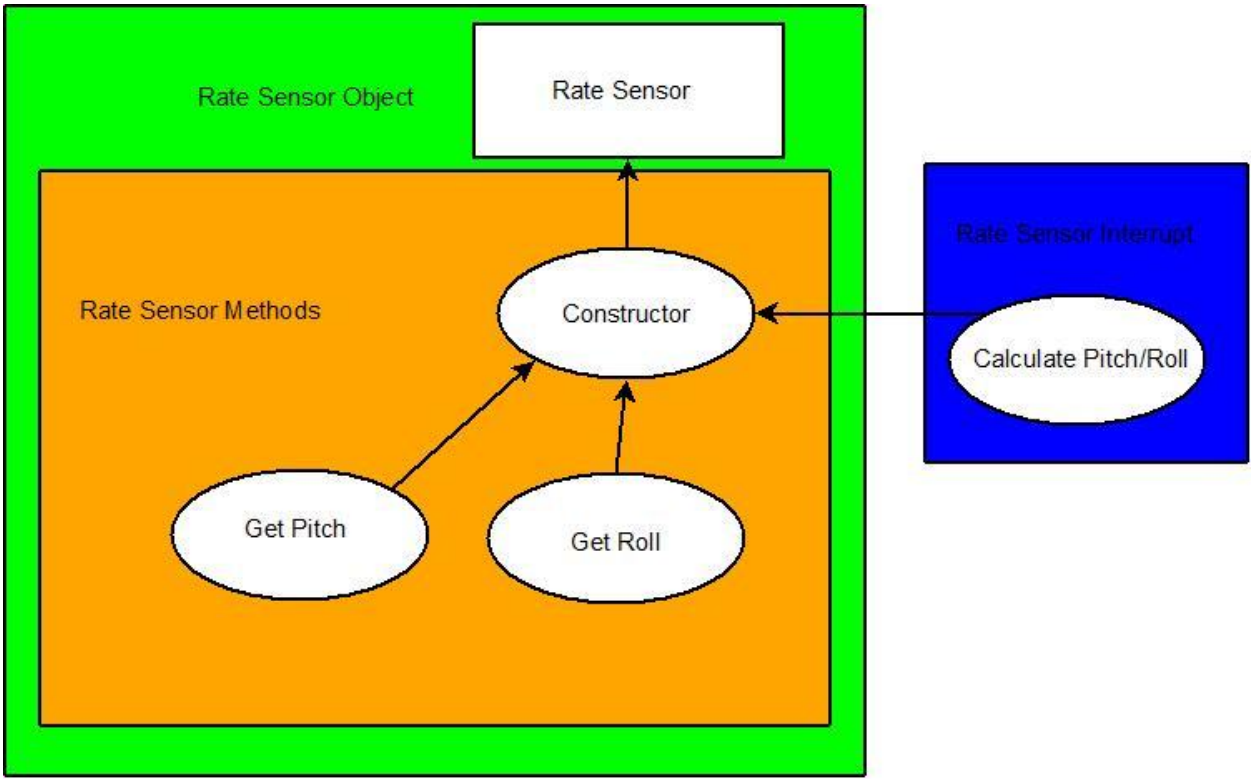


Figure 39: Rate Sensor Software Diagram

The interrupt reads the registers of the A/D converter that are associated with the pitch and roll signals. These signals are then converted to degrees through a formula provided by the MLX90609 data sheet shown in Equation 4. The bias is the DC offset of the analog signal and the gain is specific to the type of MLX90609 purchased. The determined angular rate is then added to 250 sample running average which can be accessed through the get methods.

$$V_{OUT} = Bias + Gain * AngularRate$$

Equation 4: Formula for Relating Angular Rate to Measured Voltage

5.3.3 LCD Display

The LCD display source code is the most complex in this project because it requires the usage of a many data and control signals as well as poorly documented initialization procedure. We very briefly attempted to write the base procedures for this component ourselves before realizing the level of complexity. We turned to online groups and forums to search for C++ libraries or source code that provided the command interface with the Toshiba T6963C microcontroller featured on the Crystalfontz CFAG128128A-STI-TZ LCD display. After several days of searching we found a very well written source file that illustrated the command structure and attempted to describe the initialization procedure. These files are the T6963 (LCD Processor) header and T6963 (LCD Processor) Source found in Appendix A: Source Code.

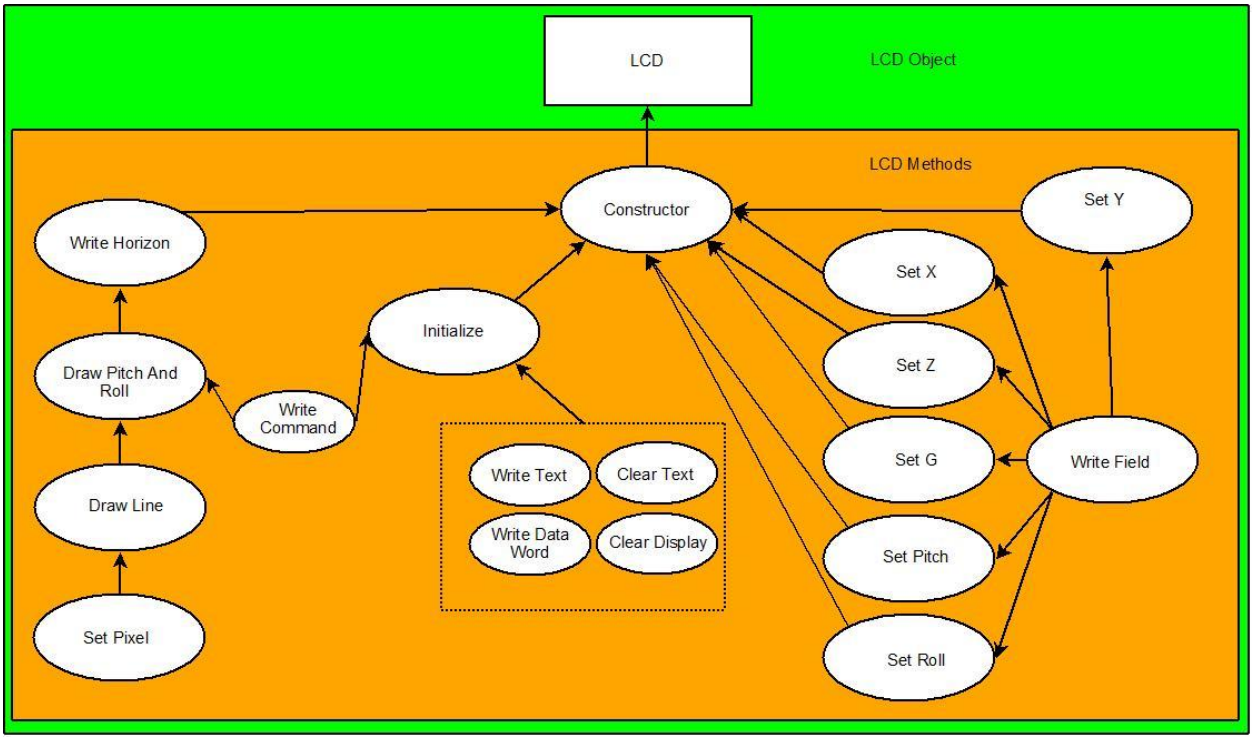


Figure 40: LCD Display Software Diagram

Figure 40 shows the software diagram for the LCD Display. The write data word and write command functions are both functions that are provided in the borrowed source code which use in the more abstract functions. Set pixel is a function that was also borrowed but is modified so that a pixel as is drawn with exclusive-or logic. This means that if a drawing a blank pixel over a pixel that is already blank will fill the pixel, and drawing a filled pixel over a filled pixel will result in a blank pixel. Any other scenario will result in a filled pixel. This was modified to prevent the draw pitch and roll function from having the redraw any lost pixels when clearing the previous line and results in less complicated drawing routine. The display contains 6 fields: X, Y, and Z correspond to the acceleration force felt in their respective directions; G is the total amount of force being experienced by the accelerometer; P and R are pitch and roll respectively. The layout of the field can be shown in Figure 41.

The set functions update these values by calling the write field function that displays the values in defined locations with defined lengths. The easy method of writing a field is to clear the previous value and then write the new one. We implemented this method in our initial program and found that it made the updating of the fields look choppy and transparent when the update frequency was high. By writing the new value directly over the old value and filling the remaining field with blank characters the algorithm was quicker and became much more readable.

Figure 41: FAADR Active LCD Display

The two orthogonal axes in the middle of the display represent the iron sights of an attitude indicator and the line drawn over the axes represents the horizon. The drawn horizon is sprite or a pre-rendered two-dimensional figure that we computed externally using the program in Appendix D – Horizon Sprite and Logic Generator Source Code. The program writes the C++ code that we use in the LCD Source for the sprites, which are stored as two one-dimensional integer arrays of a fixed length, and the logic for determining which line to draw based on the roll angle which is the write horizon function. The write horizon function also stores the necessary variables for determining which line is previously drawn in order for that line to be cleared on the next update. This gives the appearance of a rotating line when the roll sensor is manipulated.

We only generate sprites for one quadrant of the coordinate system. The draw pitch and roll function is responsible for translating the sprites over the x and/or y axis and mirroring them in the opposite quadrant in order to draw a full horizon that can represent 360 degrees. The x and y axis are represented by the CENTER_X and CENTER_Y variables defined in LCD Source.

5.3.4 SD Card

There are two additional pins that need to be accounted for when communicating with the SD Card. The Busy and Dir pin dictate what state the SD Card is in and need to be checked before performing any transfer of information through SPI bus. While this component is advertised as communicating using SPI, it is not a true SPI device because of the addition of two status pins that bring the total communication pins to 6. These two status pins are checked using the Read Dir and Read Busy functions. The Ready Dir and Ready Busy functions, as shown in Figure 42, use the read functions to determine that status of the SD Card module. The enable and disable functions are used to toggle to CS bit for this module.

The first operation that is required to write to a file on the SD Card is to send the command “ow <filename>” when filename is the name of the file that you want to open on the SD Card. The Send Open function sends this character string via the SPI bus to the module after checking the status pins. The busy and dir pins must be checked before every single character write to ensure that the module is ready to receive additional characters. When the entire character string is written, the carriage return character ‘\n’ is then sent so the module knows the command is complete. After sending the carriage return, the CS bit must also be toggled so the module knows again that the command is complete.

Once this dance is completed, a series of dummy writes are made to clock in input from the module. The module input is either a file handle that is required to perform a write to file or one of many error codes. These error codes are in the Handle SD Errors function.

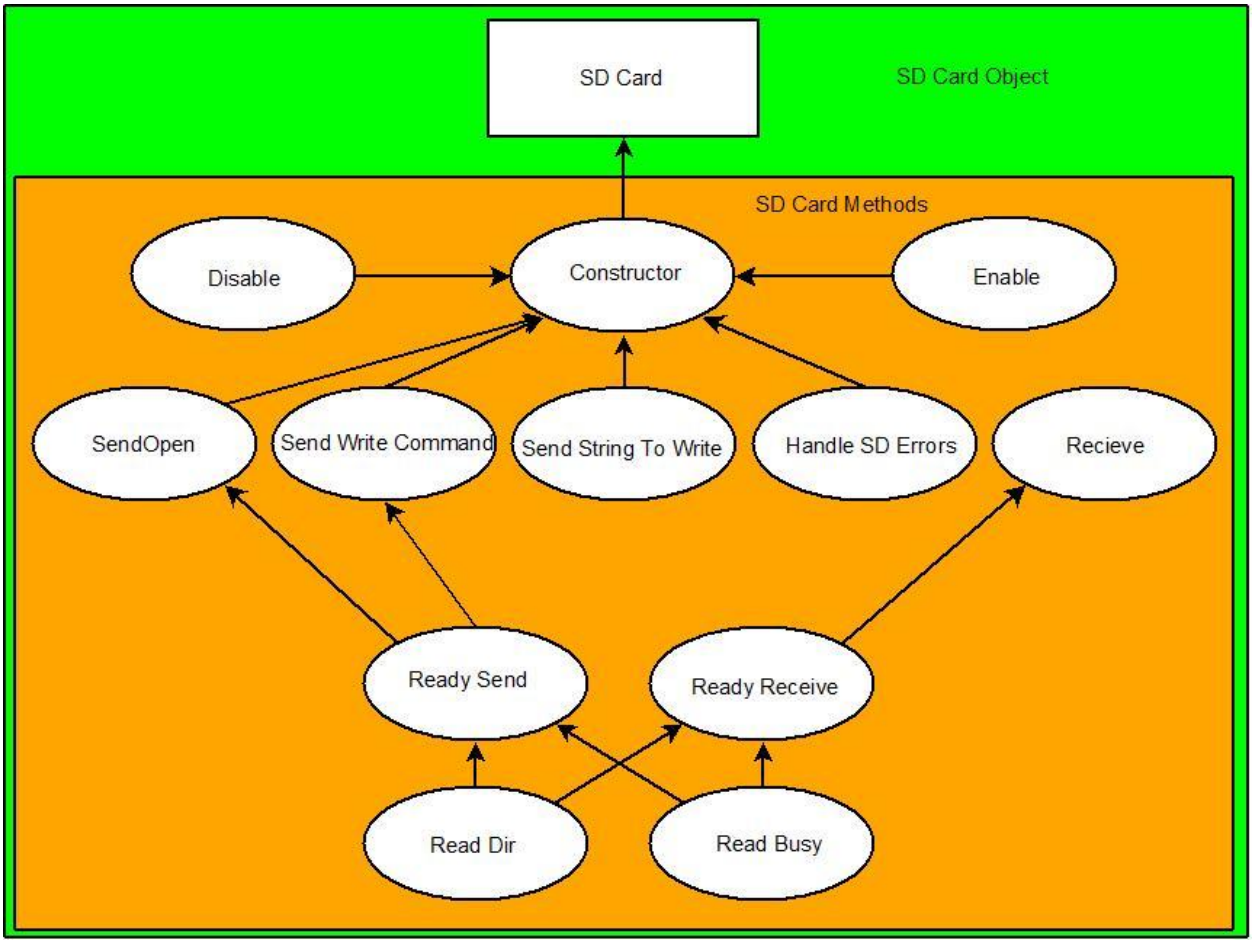


Figure 42: SD Card Software Diagram

The Send Write Command function takes in a file write command that is in the form of “w #<filehandle> <number of bytes to be written>. In writes this command to the SPI bus and if the command accepted successfully it returns with the ‘ “ ‘ character.

After the Send Write Command is successfully sent, the character string to be written is sent. Again, before every single character write the status signals have to be checked. There is no close function or

handling of multiple file handles because the FAADR will not require the need to release file handles or write to more than one file at a time. However, all file handles are closed when the FAADR initializes as a precaution.

We found the support from other users through forums was far more informational then the support we received from SparkFun.com or DOSonCHIP. One person had struggled with this module for so long that he had in effect rewritten the CD17B10 user guide to incorporate the poorly documented bugs and features. We found this particular post to be extremely helpful because it included a step by step procedure on how to successfully communicate with this device. Also included in the post was a long serious of sample code functions. These functions were very hard to follow and we found them to unusable but they did outline the procedure that we followed in our functions.

5.3.5 Main

The main function is the program that is directly run by the MSP430. It basically just initializes each component by constructing the object or the calling the initialize function. It then gathers data from the accelerometer and rate sensors, sends that data to LCD and then writes the data to the SD Card as shown in Figure 43. The SD Card write only occurs twice a second so the LCD is always updated but the write to the SD Card may be skipped if it is not necessary.

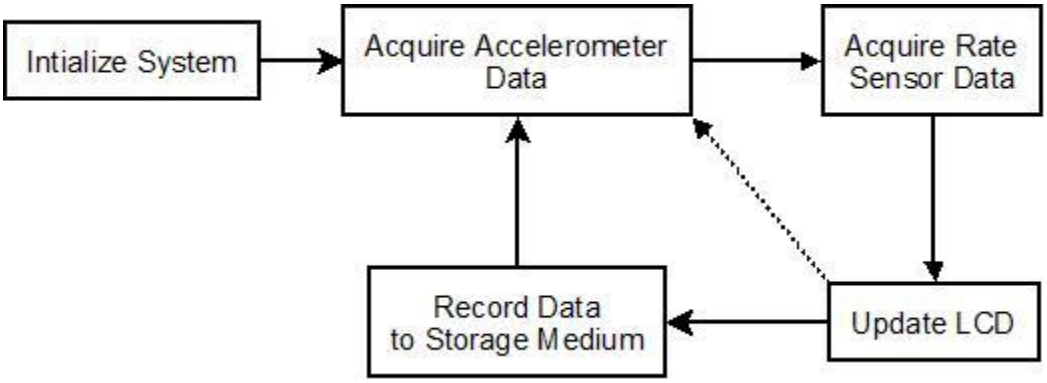


Figure 43: Internal Software System Block Diagram

5.3.6 Analysis Software

The analysis software written for the FAADR output files (source code can be found in Appendix F) is a combination of open source tools utilized to create a GUI which will retrieve a FAADR file and display graphs of the data including:

- X axis acceleration over time
- Y axis acceleration over time
- Z axis acceleration over time
- Total acceleration over time

- Total times acceleration exceeded 6 Gs

The GUI for the software utilized an open source tool known as Thinlet (ww.thinlet.com) which uses an XML file to generate an appropriate GUI based on the parameters that are given in the document. Using the Thinlet Java API the software creates an initial screen which asks the user to input a directory in which to search for relevant text files (shown in Figure 44).

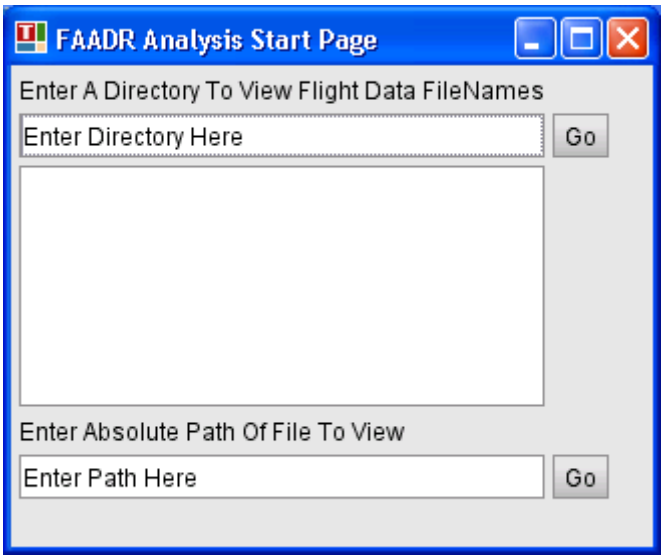


Figure 44: The initial screen of the analysis software

When the user enters a directory into the text box and clicks go the software calls on the Java Standard Library to search the given directory for all files with the '.txt' extension. All of the filenames which match the criteria are then displayed in the second text box so that the user can view the files he can choose from.

Once a file is chosen, the user then enters the absolute path of the file in the final text box and clicks the second go button. The information a user must enter before clicing the final go button is shown in Figure 45. At this point the bulk of the code for this software package is required to run. When the go button is clicked the program immediately begins parsing the comma delimited file and creating large arrays of the values for the x, y and z axis. These values are then used to create datasets for another Java based API from JfreeChart. The software creates a dataset for each of the graph types listed above and then instructs the JfreeChart to create five bitmap images for each chart.

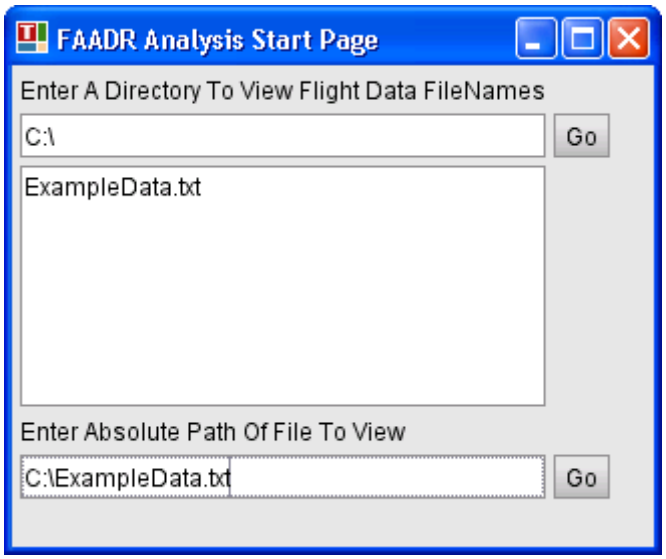


Figure 45: Software Analysis initial screen with populated values

Once the bitmap images are saved, the program passes a new xml file to Thinlet which allows the GUI to change from the initial screen, to the final results screen shown in Figure 46.

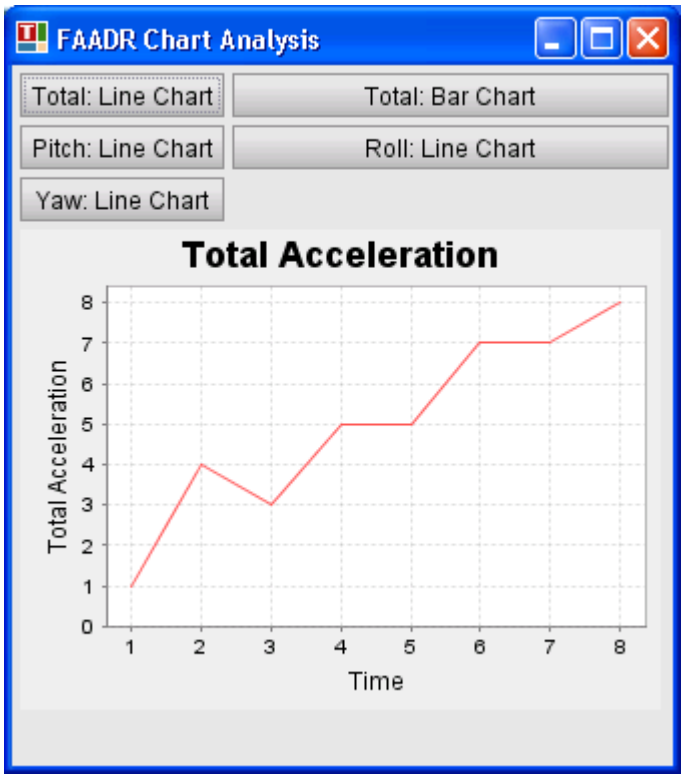


Figure 46: The final screen displayed by the analysis software which allows the user to view 5 different graphs generated from the given data

At this point the user can view all of the results of the analysis software by clicking through the different types of graphs that were created for easy and intuitive viewing. The final graphs for the ExampleData.txt are shown in Figure 46, Figure 47, Figure 48, Figure 49 and Figure 50.

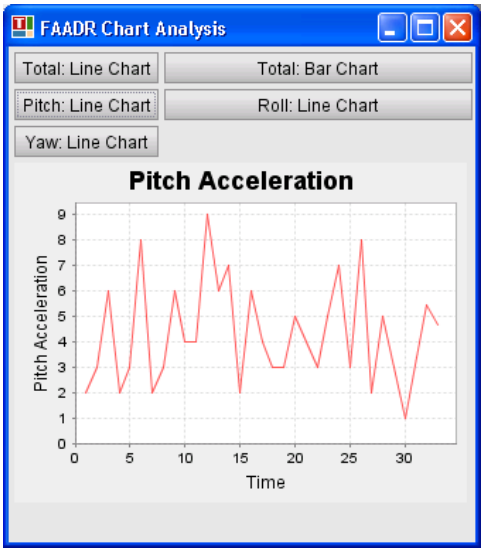


Figure 47: Graph from analysis software showing pitch acceleration over time

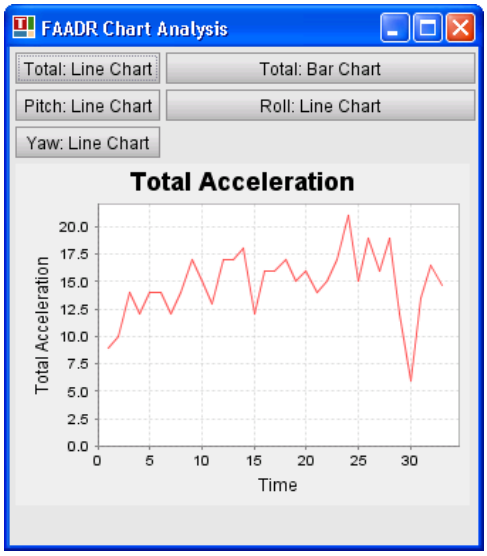


Figure 48: Graph from analysis software showing total acceleration over time

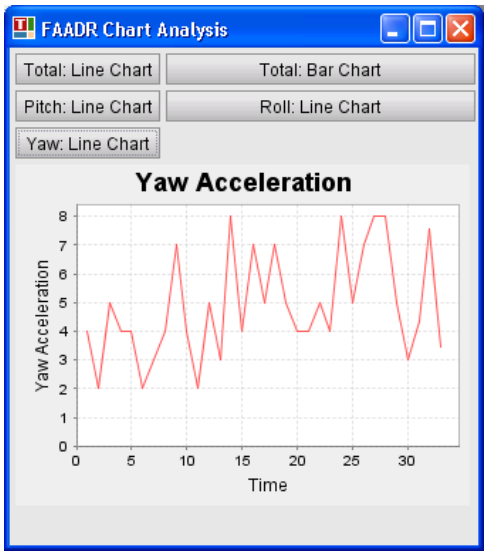


Figure 49: Graph showing the Yaw acceleration over time

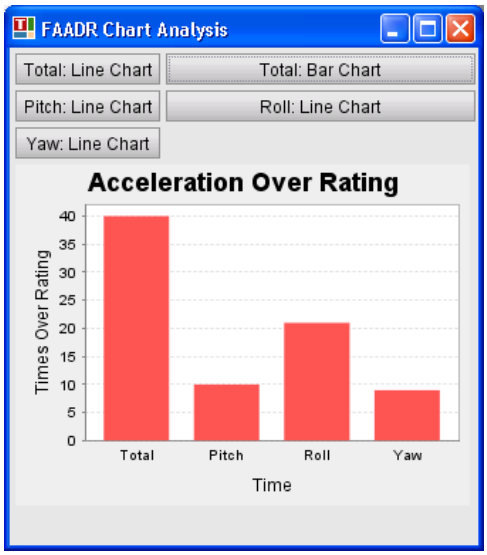


Figure 50: Graph showing the total times over 6Gs for total, pitch, roll and yaw acceleration

5.4 Analysis

The following list reiterates the requirements in Section 3 Problem Statement and states the extent which the FAADR met those requirements. Section 5.4.2 Issues, will list the issues with the final version of the FAADR.

5.4.1 Requirements

Require less than 2 W. The FAADR requires only 240mA/h during normal operation. The largest consumer of power is the LCD display which uses 150mA/h. This allows the FAADR to be run not only

from an accessory power adapter in a glider's cockpit, but also an external battery pack as long as the voltage is greater than 5 Volts.

Record tri-axial g-force measurements. The FAADR can make measurements of ±6G with a 0.05G margin of error. This range exceeds the maximum loading limit for utility class gliders which makes this instrument very robust and more accurate than required.

Record heading. While we did attempt to use an electric compass to retrieve heading during this project we failed to successfully implement one for the final product. For more information on this particular failure, see Section 5.2.4 Angular Rate Sensor MLX90609.

Record up to 12 hours of data. The result of this requirement depends entirely on the size of the SD Card provided. The FAADR records acceleration data at 1Hz. Each individual data recording is at most 20 characters which is 20 bytes, so according to Equation 5 the FAADR writes 40 bytes per second.

$$\frac{1}{s} \times 20 \text{ bytes} = 20 \frac{\text{b}}{s}$$

Equation 5: Number of bytes per second written to SD Card

$$20 \frac{b}{s} \times 60 \frac{s}{m} \times 60 \frac{m}{h} \times \frac{1}{1024} \frac{kb}{b} = 71 \frac{kb}{h}$$

Equation 6: Number of kilobytes per hour written to SD Card

$$\frac{1024 \frac{kb}{mb} \times 1024\ mb}{71 \frac{kb}{h}} = 14{,}872\ h$$

Equation 7: Hours of data recorded on 1 GB SD Card

Equation 6 and Equation 7 expand the units of time and memory and show that the number of hours we can record on a 1 GB SD Card is almost 15,000 hours. This far surpasses our requirement. The stipulation is that we do not support multiple files. The FAADR only writes to one file, "LOG.TXT." The SD Card must be formatted to permanently delete that data in the file or the FAADR will append the LOG.TXT file the next time the recording begins. For information about the SD Card, see Section 5.3.4 SD Card.

20 bytes is the longest string that we would ever write which would mean that every acceleration vector's magnitude was greater than or equal to 1000 mG and was negative. This condition requires 5 characters a piece which brings to total to 15 characters. With the addition of separation characters, the total characters to be written to the SD Card at 2Hz are about 20 characters.

Display data on graphical LCD. The FAADR does not display maximum G experienced during flight, but it does display all the vectors from the accelerometer, the current G force, pitch and roll. The artificial horizon is also displayed on the screen.

Display current heading. Heading is not displayed on the FAADR because of the series of debacles we experienced with electric compasses. For information see Section 5.2.4 Angular Rate Sensor MLX90609.

Reset display values. We have implemented two push button switches to control the user interface. During operation, by activating one of the push buttons the angular rate sensor integration is reset for pitch and roll. This is useful because if the rate sensors drift at all during flight, the pilot can reset the values we he experiences straight and level flight. This will effectively zero the rate sensors and allow them to resume normal operation.

Include analysis software that runs a PDA and PC. The analysis software is a Windows executable written in Java, so any PDA with a Windows operating system and Java installed should be able to run it. The UI for the software is designed to run on the PDA but we never tested this. It does work on a PC with a Windows operating system. For more information, see Section 5.3.6 Analysis Software.

5.4.2 Issues

When tying all of the components together in software, we found that the FAADR was operating at a noticeably slow rate. The SD Card was being written to at the correct frequency but the LCD display drawing routine was experiencing noticeable lag. We attempted many software tweaks including using the auto write-up function offered by the T6963C, increasing the baud rate for the SD Card and the accelerometer, and decreasing the sampling frequency of the rate sensors.

The auto write-up function cannot be used for non-sequential memory addresses, which makes it useless for LCD drawing routine. The baud rate for the SD Card could not be increased because of the mandatory wait times needed in order for the unit to work. Since the accelerometer and the SD Card are on the same SPI port, we could not increase the speed of transfer from the accelerometer. However, the communication with the accelerometer was not a huge contributor to the microprocessors load.

In order to obtain a decent rate sensor reading, we required a high sampling frequency. In order to make a correct interpretation of the signal we required some sort of average. We used a block average and found that this made the signal more accurate but still very prone to error. We averaged all the signals that we received in a second and added them to a running total. This increased the accuracy but we still experienced some error. We then used double values instead of integer values which again increased the accuracy, but we still experienced noticeable error. Given the available processing power, we assumed that this was the best we could do in terms of precision.

Using the enclosure as a make shift compass we preceded to modify the measurement weights to adjust the signal to be about 90, 180, 270 or 0 degrees by rotating the box accordingly. We adjusted for pitch only and found then when we added roll it no longer was correct. Then we adjusted one axis at a time, but left both in to be calculated. While pitch seemed to be within 3 degrees of its expected value, roll was not. These weighted constants are dependent on temperature which makes assigning static constants relatively pointless.

In truth, the attitude indicator does not work well. We have adjusted it to perform as well as possible but there are many sources of error that we cannot control such as the analog/digital conversion, perfect orthogonal mounting, and mathematical precision during calculation. The other truth is that if the EZ-Compass 3 had not malfunctioned then we would not have had this issue. However, if we had included the EZ-Compass 3 as part of the final product then the cost of the FAADR would have more than doubled. In this case, we have traded precision and accuracy for cost.

To make the attitude indicator more usable, the SD Card write frequency needed to be adjusted from a more optimum frequency of 2Hz to 1Hz. The change was necessary to preserve the accuracy and speed of the attitude indicator display. When attempting to write at 2Hz the interrupts began thrashing and the system entered a state from which it could not recover.

5.5 Summary

This section showed how we implemented each component of the FAADR physically and how we used software to get all the components working according to the requirements. In the Results section we showed how well the final product met the requirements set at the beginning of the project.

6 Conclusion

6.1 Project Summary

The purpose of this project was to create an inexpensive and low power attitude indicator and acceleration data recorder. At the conclusion of this project, the FAADR was able to display its orientation on an LCD screen and record the acceleration experienced on an SD Card. The software that we created allows the pilot to interpret the post-flight data.

Figure 51: FAADR Main Enclosure with Accessory Power Socket Adapter and Ethernet cable

While the attitude indicator is not as usable, the acceleration recording is on par with the commercial instruments. The final product cost for the FAADR is about $300. This makes cost about a third as much as its' commercial counterparts.

Figure 52: FAADR Sensor Enclosure with Ethernet cable

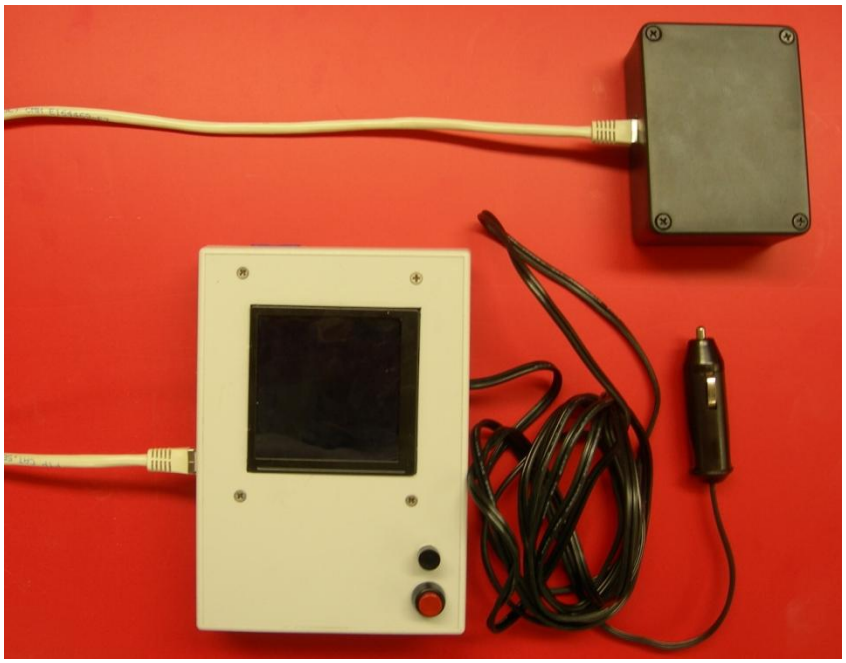


Figure 53: FAADR Final Product

6.2 Future Work

In order to make the attitude indicator functionally usable, we recommend that an MSP430 with a greater clock frequency, such as 5 or 10 MHz, be used. This would allow the system to sample the angular rate sensors more often and would reduce the amount of lag observed when drawing to the LCD

screen. With this simple change the FAADR could then accurately sample and display attitude data of a glider and become a commercially viable alternative to current products.

Works Cited

[1] Eismin, Aircraft Electricity and Electronics, Career Education, 1994.

[2] "Horizon," [Online], September 14, 199, Available: http://www.sgsim.com/instruments/horizon.htm

[3] Langewiesche, Stick and Rudder: An Explanation of the Art of Flying, 1 ed.,McGraw-Hill Professional, 1990.

[4] Monteith , Simple Aerodynamics and The Airplane, 3 ed. , The Ronald Press Company, 1929.

[5] Perkins, Airplane Performance and Stability Control, 1 ed. , Wiley, 1949.

[6] Sears, Airplane and its Components, New York: John Wiley and Sons, 1942.

[7] White , Gliding and Soaring: An Introduction to Motorless Flight, Whittlesey House, 1931.

[8] National Archive and Records Administration, "Electronic Code of Federal Regulations: Title 14, Part 23, Subpart C, Subsection 23, e-CFR Data is current as of September 20, 2007". [Online]. Available: http://ecfr.gpoaccess.gov/cgi/t/text/text-idx?type=simple;c=ecfr;cc=ecfr;sid=cae60a72b5461f42cac90f10b6b0e9eb;region=DIV1;q1=utility%20category;rgn=div8;view=text;idno=14;node=14%3A1.0.1.3.10.3.70.10. [Accessed: September 30, 2007].

[9] Honeywell, "Digital Compass Solution HMC6352" http://www.sprkfun.com/datasheets/Components/HMC6352.pdf

[10] Anders Persson, "How Do We Understand the Coriolis Force," Atmospheric and Oceanographic Sciences Library," European Centre for Medium-Range Weather Forecasts, Reading, Berkshire, United Kingdom. [Online]. Available: http://www.aos.princeton.edu/WWWPUBLIC/gkv/history/Persson98.pdf [Accessed: April 11, 2008].

[11] VTI Technologies, "SCA3000 3-Axis Series, Product Family Specification". [PDF]. Available: http://www.vti.fi/midcom-serveattachmentguid-88f74b45c976228bbbc1dd5c542ac085/sca3000_accelerometer_product_family_specification_8257300a.05.pdf

[12] Melexis Microelectric Integrated Systems, "MLX90609 Angular Rate Sensor", [PDF]. Available: http://www.melexis.com/Sensor_ICs_Inertia/General/MLX90609_582.aspx

[13] http://www.wingsandwheels.com/images/AttitudeTurn.jpg

[14] http://mtp.jpl.nasa.gov/notes/pointing/Aircraft_Attitude2.png

[15] http://www.dkimages.com/discover/Home/Technology/Transportation/Aircraft-and-Aviation/Aircraft-Components/Instruments/Instruments-10.html

[16] Wings and Wheels Soaring Supplies, "MORE INSTRUMENTS AND INSTALLATION SUPPLIES," *wingsandwheels.com*, HORIZON/COMPASS Attitude/Heading Indicators. [Online]. Available: http://www.wingsandwheels.com/page23.htm [Accessed: October 05, 2007].

[17] http://www.mccullagh.demon.co.uk/update2-3.htm

[18] Berkeley, “Secondary Transducers”, Berkeley Data is current as of January 2000 [Online]. Available: http://bits.me.berkeley.edu/beam/acc_2b.html. [Accessed: October 7, 2007].

[19] “GPS, DGPS, and Backup Systems”, Data is current as of September 4, 1998. [Online]. Available: http://lapierre.jammys.net/masters/index.html. [Accessed: October 7, 2007].

[20] PNI, “Discrete Circuit for Magneto-Inductive Sensors”, PNI Data is current as of December 2003. [Online]. Available: https://www.pnicorp.com/downloadResource/c1d/manuals/67/Ap+Note+Discrete+Circuit+for+MagnetoInductive+Sensors.pdf. [Accessed: October 7, 2007].

[21] Fixed Wing Flight Training, “Attitude Indicators” [Online]. Available.: http://www.pilotfriend.com/training/flight_training/fxd_wing/attitude.htm [Accessed: October 7, 2007]

[22] “Attitude Indicator”, September 7, 2007. Available: http://en.wikipedia.org/wiki/Attitude_indicator. [Accessed: October 7, 2007].

Appendix A: Source Code

Accelerometer Header

```
#ifndef ACCELEROMETER_H_
#define ACCELEROMETER_H_
#include "io430x14x1.h"
#include "LCD.h"
#include "Compass.h"


struct accelData {
  char x [4];
  char y [4];
  char z [4];
  int xNum;
  int yNum;
  int zNum;
  unsigned long gNum;
  char g [6];
};
class Accelerometer
{
public:
      Accelerometer();
        unsigned int read_register(unsigned int Address);
        unsigned int read_8bit_register(unsigned int Address);
        void enable_buffer();
        void read(struct accelData* data);
        void demo(Compass comp, LCD lcd) ;
        int format(unsigned int i, char * n);
        char * itoa(int val, char *buf, int radix);
        void  xtoa(unsigned  long  val,  char  *buf,  unsigned  radix,  int
negative);
};

#endif /*ACCELEROMETER_H_*/
```

Accelerometer Source

```
#include "Accelerometer.h"
#include <math.h>

Accelerometer::Accelerometer()
{
//  enable_buffer();
  P3SEL |= BIT0 + BIT1 + BIT2 + BIT3;                       // Setup P3
for SPI mode
```

```
  U0CTL = CHAR + SYNC + MM + SWRST;                 // 8-bit, SPI, Master
  U0TCTL = CKPH + SSEL0 + STC;          // Polarity, SMCLK, 3-wire, removed
ssel1!! put it back later changed also changed ckph ckpl
  U0BR0 = 0x03;                              // SPICLK = ACLK/3 = 32K/9600 = 3
  U0BR1 = 0x00;
  U0MCTL = 0x00;
  ME1 = USPIE0;                              // Module enable
  U0CTL &= ~SWRST;                           // SPI enable
  IE1 |= URXIE0 + UTXIE0;                    // RX and TX interrupt enable
}
/* This example reads SCA3000 X acceleration registers MSB and LSB.
* The register address of X channel is 0x05 (MSB).
* NOTE: The command byte of SPI bus is :
* (MSB) A A A A A A A RW 0 (LSB)
* | | | *- always zero
* | | *--- Read / Write Bit
* \-----------*------ Register address, 0x05 at this example
*
* Because of the real byte sent to SPI bus must be calculated:
* BYTE = (Address * 4) + RW*2;
* For Read operation RW = 0 and for Write operation RW = 1
* 'Address *4' shifts the bit pattern to left by 2 and 'RW*2' by 1.
* BYTE = 0x05*4 + 0;
* BYTE = 0x14;
* This calculation is done inside the function.
*
* Usage of the following function
*
* int Xacc;
*
* Xacc = SCA3000_read_16bit(0x05);
*/
// Function takes in the 8-bit register address and returns 16-bit register
value.
unsigned int Accelerometer::read_register(unsigned int Address) {
  int result = 0;
  int x = 0;
  Address = Address << 2;    // RW bit is set to zero by shifting the bit
pattern to left by 2
  P2DIR |= BIT5;
  P2OUT &= ~BIT5;           // Turn Accelerometer CS bit to ON
  U0TXBUF = Address;        // Write command to SPI bus
  while(!(IFG1 & UTXIFG0)); // Wait till command byte is written
  for(volatile int i = 0; i<50; i++);
  U0TXBUF = 0x00;            // Write dummy byte to line in order to generate
SPI clocks for data read
  while(!(IFG1 & UTXIFG0));
  for(volatile int i = 0; i<50; i++);
  while(!(IFG1 & URXIFG0)); // wait for response
  result = U0RXBUF;         // Get MSB of the result
  result = result << 8;     // Shift the MSB of the result to left by 8
   for(volatile int i = 0; i<50; i++);
  U0TXBUF = 0x00;            // Write dummy byte to line in order to generate
SPI clocks for data read
   for(volatile int i = 0; i<50; i++);
  while(!(IFG1 & UTXIFG0));
  while(!(IFG1 & URXIFG0)); // wait for response
```

```
  x = U0RXBUF;  // Get LSB of the
  result |= x;              // Place LSB of register into LSB of result
  P2OUT |= BIT5;            // Set CSB to one
  return result;
}

unsigned int Accelerometer::read_8bit_register(unsigned int Address) {
  char result = 0;
  Address = Address << 2;    // RW bit is set to zero by shifting the bit
pattern to left by 2
  P2DIR |= BIT5;
  P2OUT &= ~BIT5;           // Turn Accelerometer CS bit to ON
  U0TXBUF = Address;        // Write command to SPI bus
  while(!(IFG1 & UTXIFG0)); // Wait till command byte is written
  for(volatile int i = 0; i<100; i++);
  U0TXBUF = 0x00;            // Write dummy byte to line in order to generate
SPI clocks for data read
  while(!(IFG1 & UTXIFG0));
  while(!(IFG1 & URXIFG0)); // wait for response
  result = U0RXBUF;         // Get MSB of the result
            // Place LSB of register into LSB of result
  P2OUT |= BIT5;            // Set CSB to one
  return result;
}

/* This example reads SCA3000 MODE register content.
* The MODE register address is 0x14.
* NOTE: The command byte of SPI bus is :
* (MSB) A A A A A A A RW 0 (LSB)
* | | | *- allways zero
* | | *--- Read / Write Bit
* \-----------*------ Register address, 0x14 at this example
*
* Because of that the real byte send to SPI bus must be calculated:
* BYTE = (Address * 4) + RW*2;
* For Read operation RW = 0 and for Write operation RW = 1
* 'Address *4' shifts the bit pattern to left by 2 and 'RW*2' by 1.
* BYTE = 0x14*4 + 0;
* BYTE = 0x50;
* This calculation is done inside the function.
*
* Use of the following function:
*
* SCA3000_read_mode();
*
* This example has been written for an Atmel ATmega168 processor in a GCC
environment.
* Other processors or environments might need different names ports or SPI
device names.
* SPDR - is the read/ write data to/ from SPI bus register
* (see ATmega168 document (doc2545.pdf) pages 159 - 167 for more information
*/
// Read SCA3000 MODE register and return its value.
void Accelerometer::enable_buffer() {
  unsigned int reg_write = 0x14*4 + 2*1;
  int mode;
  mode = 0x80;
```

```
  P2DIR |= BIT5;
  P2OUT &= ~BIT5;             // Turn Accelerometer CS bit to ON
  while(!(IFG1 & UTXIFG0));   // wait for tx buffer empty
  U0TXBUF = reg_write;        // Write command to SPI bus
  while(!(IFG1 & UTXIFG0));   // wait for tx buffer empty
  U0TXBUF = mode;             // Write command to SPI bus
  while(!(IFG1 & UTXIFG0));   // wait for tx buffer empty
  P2OUT |= BIT5;              // Set CSB to one, Accelerometer off
}


int Accelerometer::format(unsigned int i, char * n) {
    signed int l = i;
    if(i >= 0x8000) {         //Test if number is negative
      l = -(0x10000 - l);     //Apply sign and convert from two's complement
    }
    l = l / 8;                //Remove the three low DNC bits
    l = l * 2;                        //Equivalent to dividing by counts and
multiplying by 1000
    itoa(l,n,10);  //Convert number to character
    return l;
}

void Accelerometer::read(struct accelData* data) {
    data->xNum = read_register(0x05);
    format(data->xNum, data->x);
    data->yNum = read_register(0x07);
    format(data->yNum, data->y);
    data->zNum = read_register(0x09);
    format(data->zNum, data->z);
    data->gNum = sqrt(pow((double) data->xNum, 2) + pow((double) data->yNum,
2) + pow((double) data->zNum, 2));
    format(data->gNum, data->g);
}

void Accelerometer::xtoa(unsigned long val, char *buf, unsigned radix, int
negative) {
      char *p;
      char *firstdig;
      char temp;
      unsigned digval;

      p = buf;

      if (negative) {
            // Negative, so output '-' and negate
            *p++ = '-';
            val = (unsigned long)(-(long) val);
      }

      // Save pointer to first digit
      firstdig = p;

      do {
            digval = (unsigned) (val % radix);
            val /= radix;
```

```
            // Convert to ascii and store
            if (digval > 9)
                  *p++ = (char) (digval - 10 + 'a');
            else
                  *p++ = (char) (digval + '0');
      } while (val > 0);

      // We now have the digit of the number in the buffer, but in reverse
      // order.  Thus we reverse them now.

      *p-- = '\0';
      do {
            temp = *p;
            *p = *firstdig;
            *firstdig = temp;
            p--;
            firstdig++;
      } while (firstdig < p);
}

char * Accelerometer::itoa(int val, char *buf, int radix) {
      if (radix == 10 && val < 0)
            xtoa((unsigned long) val, buf, radix, 1);
      else
            xtoa((unsigned long)(unsigned int) val, buf, radix, 0);

      return buf;
}
```

Rate Sensor Header

```
#ifndef RATESENSOR_H_
#define RATESENSOR_H_

struct orientationData {
      int pitch;
      int roll;
      char pitchChar [5];
      char rollChar [5];
};
class rateSensor {
      public:
            rateSensor();                 //Inits the rateSensors and gets
the initial
            void reset();                 //Clears pitch and roll and gets
new offset
            int getPitch();               //Returns the calculated Pitch
value
            int getRoll();                //Returns the calculated Roll
values
            int checkPitch();       //Gets the Pitch Value from the sensor
```

```
            int checkRoll();        //Gets the Roll value from the sensor
            void calculatePitch();  //calculates   the   real   Pitch   using
checkPitch();
            void calculateRoll();   //Calculates   the   real   Roll   using
checkRoll();
                void Timer_A();
};

#endif /*RATESENSOR_H_*/
```

Rate Sensor Source

```
#include "rateSensor.h"
#include  <msp430x14x.h>
#include <math.h>

static int pitch;
static int roll;
static int offset;

rateSensor::rateSensor(){
      P6SEL = 0x0F;                             // Enable A/D channel inputs
      ADC12CTL0  =  ADC12ON+MSC+SHT0_2;               // Turn on ADC12, set
sampling time
      ADC12CTL1 = SHP+CONSEQ_1;                  // Use sampling timer, single
sequence
      ADC12MCTL0 = INCH_0;                      // ref+=AVcc, channel = A0
      ADC12MCTL1 = INCH_1;                      // ref+=AVcc, channel = A1
      ADC12CTL0 |= ENC;                         // Enable conversions
        pitch = 0;
      roll = 0;
      offset = 0;
        ADC12CTL0 |= ADC12SC;
        P1DIR |= 0x01;                            // P1.0 output
        CCTL0 = CCIE;                             // CCR0 interrupt enabled
        CCR0 = 50000;
        TACTL = TASSEL_2 + MC_2;                  // SMCLK, contmode
        offset = checkPitch();
        _BIS_SR(CPUOFF + GIE);                  // Enter LPM0 w/ interrupt

}

int rateSensor::getPitch(){
      return pitch;
}

int rateSensor::getRoll(){
      return roll;
}
```

```
int rateSensor::checkPitch(){
      return ADC12MEM0;
}

int rateSensor::checkRoll(){
      return ADC12MEM1;
}

void rateSensor::calculatePitch(){
      int result = checkPitch();
      if(result>offset){
          if((result-offset)>300)
            pitch += (result-offset)/100;
        }
      else if(result<offset){
           if((offset-result)>300)
            pitch -= (offset-result)/100;
        }
}

void rateSensor::calculateRoll(){
      int result = checkRoll();
      if(result>offset)
            pitch += (result-offset)/100;
      else if(result<offset)
            pitch -= (offset-result)/100;
}

// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  P1OUT ^= 0x01;                            // Toggle P1.0
  CCR0 += 10000;                            // Add Offset to CCR0
  int result = ADC12MEM0;
      if(result>offset){
          if((result-offset)>300)
            pitch += (result-offset)/100;
        }
      else if(result<offset){
           if((offset-result)>300)
            pitch -= (offset-result)/100;
        }
  _BIC_SR_IRQ(CPUOFF);                     // Clear CPUOFF bit from 0(SR)
}
```

SD Card Header

```
#ifndef SDCARD_H_
#define SDCARD_H_
```

```
#include "t6963c.h"

class SDCard
{
public:
	SDCard();
	  //Used to signify if a file is open at the current time
	int handle;
	/*PRE:	This method takes in an error code in the form of
	 *		an int
	 *POST:	This method returns the corresponding error string
	 *		from the entered error code
	 */
	char *handleSDErrors(int errorCode);

	/*PRE:	This method takes in a string to be written to
	 *		a file that is opened
	 *POST:	This method will write the given string to the end of
	 *		the file that must be previously opened.  Will return an
	 *		errorCode int if no file was previously opened.
	 */
	int write(char *writeString);

	/*PRE:	This method takes in a path fo a file to be open
	 *		in string format
	 *POST:	The method opens the file specified by the path
	 *		for reading
	 *NOTE: Only one file may be open at a time.  The handle for
	 *		the file connection is stored in the instance variable
	 *		handle
	 */
	int openFile(char *pathToFile);


	/*PRE: This method will close the file that has been previously
	 *		opened
	 *POST: This method closes the file and returns an error code
	 *		upon error including if no file is open.
	 */
	int close();
	int read_dir();
	int read_busy();
	int readySend();
	char send_open(char * command);
	  void record(char handle, struct compassData * cData, struct accelData
* aData);
	  char send_writeCommand(char * command);
	  char send_StringToWrite(char * command);
	void wait();
	char receive();
	void enable();
	void disable();
	void reset();
	int readyReceive();
	  void waitBusy();
};
```

```
#endif /*SDCARD_H_*/
```

SD Card Source

```
#include "SDCard.h"
#include "Accelerometer.h"
#include "Compass.h"
#include <io430x14x1.h>
#include <string.h>

SDCard::SDCard() {
	P2DIR &= ~BIT3;
	P2DIR &= ~BIT1;
	P2DIR |= (BIT2+BIT4);
	P2OUT |= BIT2;
}

/*
 * Get the status of the dir pin
 * 1 = busy
 * 0 = ready
 */
int SDCard::read_dir() {
	int i = P2IN & BIT3;
	if (i>0) {
		return 1;
	} else
		return 0;
}

/**
 * Get the status of the busy pin
 */
int SDCard::read_busy() {
	int i = P2IN & BIT1;
	if (i>0) {
		return 1;
	} else
		return 0;
}

/*
 * Returns 1 if ready, 0 is not ready
 */
int SDCard::readySend() {
	if ((read_dir()==0) && (read_busy()==0)) {
		while (true) {
			wait();
			enable();
			while (!(IFG1 & UTXIFG0))
```

```
                        ;// wait for tx buffer empty
                  U0TXBUF = 0x00;
                  while (!(IFG1 & URXIFG0))
                        ;
                  char c = U0RXBUF;
                  if (c == '>') {
                        break;
                  } else {
                        disable();
                  }
            }
            return 1;

      } else
            return 0;
}

int SDCard::readyReceive() {
      if ((read_dir()==1) && (read_busy()==0))
            return 1;
      else
            return 0;
}

void SDCard::waitBusy() {
      while (read_busy()==1)
            ;
      return;
}

/**
 * Send a string to the card
 */
char SDCard::send_open(char * command) {
      reset();
      wait();
      enable();
      while (!readySend());
      for (unsigned int i=0; command[i] != '\0'; i++) {
            waitBusy();
            while (!(IFG1 & UTXIFG0));
            wait();
            U0TXBUF = command[i];
            while (!(IFG1 & UTXIFG0));
            wait();
      }
      enable();
      wait();
      while (!(IFG1 & UTXIFG0));
      U0TXBUF = 0x0D;
      while (!(IFG1 & UTXIFG0));
      return receive();
}

char SDCard::send_StringToWrite(char * command) {
      wait();
      for (unsigned int i=0; command[i] != '\0'; i++) {
```

```
            waitBusy();
            while (!(IFG1 & UTXIFG0));
            wait();
            U0TXBUF = command[i];
            while (!(IFG1 & UTXIFG0));
            wait();
      }
      wait();
      return receive();
}

void SDCard::record(char handle, struct compassData * cData, struct accelData
* aData) {
    int bytes = sizeof(aData->x) + sizeof(aData->y) + sizeof(aData->z);

}

char SDCard::send_writeCommand(char * command) {
      wait();
      while (!readySend());
      for (unsigned int i=0; command[i] != '\0'; i++) {
            waitBusy();
            while (!(IFG1 & UTXIFG0));
            wait();
            U0TXBUF = command[i];
            while (!(IFG1 & UTXIFG0));
            wait();
      }
      wait();
      while (!(IFG1 & UTXIFG0));
      U0TXBUF = 0x0D;
      while (!(IFG1 & UTXIFG0));
      return receive();
}

/*
 * Wait for the prescribed time between commands
 */
void SDCard::wait() {
      for (volatile int i = 0; i<300; i++);
}

/*
 * Gets something from SD card
 */
char SDCard::receive() {
        wait();
        int c = 0x00;
        while(c==0x0D || c==0x00) {
          U0TXBUF = 0x00;
          wait();
          c = U0RXBUF;
        }
      return U0RXBUF;
}

void SDCard::reset() {
```

```
	P2OUT &= ~BIT2;
	wait();
	wait();
	disable();
	wait();
	P2OUT |= BIT2;
	wait();
	enable();
}

void SDCard::enable() {
	wait();
	P2OUT &= ~BIT4;
	wait();
}

void SDCard::disable() {
	wait();
	P2OUT |= BIT4;
	wait();
}
/*PRE:		This method takes in an error code in the form of
 *		an int
 *POST:		This method returns the corresponding error string
 *		from the entered error code
 */
char * SDCard::handleSDErrors(int errorCode) {
	if (errorCode>0)
		return "SD Card Ready";
	else if (errorCode<=-43)
		return "FFD5 to FF00 card physical read/write error";
	else {
		switch (errorCode) {
		case -1:
			return "reserved";
		case -2:
			return "card not detected";
		case -3:
			return "could not be initialized";
		case -4:
			return "could not have block length set";
		case -5:
			return "card voltage not in range: 2.7V - 3.6V";
		case -6:
			return "card current draw above: 80mA";
		case -7:
			return "card read error";
		case -8:
			return "reserved";
		case -9:
			return "input buffer overrun error";
		case -10:
			return "invalid command";
		case -11:
			return "syntax error";
		case -12:
			return "parameter out of range";
```

```
            case -13:
                  return "end-of-file error";
            case -14:
                  return "long filenames not supported";
            case -15:
                  return "FAT12 file system not supported, please reformat";
            case -16:
                  return "incompatible file system";
            case -17:
                  return "drive does not exist/invalid drive";
            case -18:
                  return  "invalid  directory  name/not  a  directory/filename
error";
            case -19:
                  return "file/directory not found; entry does not exist in
specified dir";
            case -20:
                  return "type cannot be opened error (no hidden, system, vol
label, subdir)";
            case -21:
                  return "file already opened";
            case -22:
                  return "card changed with files open error";
            case -23:
                  return "reserved";
            case -24:
                  return "reserved";
            case -25:
                  return "invalid handle/handle out of range";
            case -26:
                  return "no handles available/no more files can be opened";
            case -27:
                  return "handle previously assigned";
            case -28:
                  return "handle not assigned|not open for this operation";
            case -29:
                  return "not opened for read operation error";
            case -30:
                  return "reserved";
            case -31:
                  return "reserved";
            case -32:
                  return "reserved";
            case -33:
                  return "sector out of range write error";
            case -34:
                  return "disk full error";
            case -35:
                  return "root directory full error (FAT16 only)";
            case -36:
                  return "lost cluster(s) error";
            case -37:
                  return  "corrupt  filesystem  error  (write  error  caused
malformed file structure";
            case -38:
                  return "read-only error/access is denied";
            case -39:
```

```
                return "not opened for write operation error";
            case -40:
                return "duplicate name error";
            case -41:
                return "directory not empty";
            case -42:
                return "open file cannot be deleted error";
            }
        }
        return "fail";
}

/*PRE:      This method takes in a string to be written to
 *          a file that is opened
 *POST:     This method will write the given string to the end of
 *          the file that must be previously opened.  Will return an
 *          errorCode int if no file was previously opened.
 */
int SDCard::write(char *writeString) {
        if (handle==-1)
            return -19;
        //TODO: insert code to write a string
        return -1;
}

/*PRE: This method will close the file that has been previously
 *          opened
 *POST: This method closes the file and returns an error code
 *          upon error including if no file is open.
 */
int SDCard::close() {
        if (handle==-1) {
            return -19;
        } else {
            //TODO close file using command
            return -1;
        }
}
```

LCD Header

```
#ifndef LCD_H_
#define LCD_H_
#include "t6963c.h"
#include "Compass.h"
#define CROSSHAIR_LENGTH 30
#define CENTER_X 64
#define CENTER_Y 64
class LCD {
public:
        LCD();
```

```
int previousSign;
int previousPitch;
int previousRoll;
int* previousYLine;
int* previousXLine;
void writeNum();
void writeText();
void writeLine();
void drawHorizon(int angle);
void writeBaseTemplate();
void resetScreen();
void writeSDMessage(char *message);
bool isReady();
int busy();
void read_command();
void end();
void init();
void write_command(char a);
void command_write();
void data_read();
void data_write();
void write_dataword(int a);
void write_data(char a);
void clear_text(int x, int y, int w);
void write_text(int x, int y, char *g);
void write_field(int x, int y, char *g);
void clear_display(void);
void fill_display(void);
void reset(void);
void clear_all_text();
void clear(int x, int y, int h, int w);
int clear_line(int x1, int y1, int x2, int y2);
int lcd_line(int X1, int Y1, int X2, int Y2);
void write_horizontal_line(int x1, int y1, int x2);
unsigned char lcd_read_data(void);
unsigned char lcd_bit_to_byte(unsigned char bit);
void lcd_set_pixel(unsigned char x, unsigned char y);
void lcd_clear_pixel(unsigned char x, unsigned char y);
void setHeading(char *g);
void clearHeading();
void setX(char *g);
void setY(char *g);
void setZ(char *g);
void setRoll(char *g);
void setPitch(char *g);
void setG(char *g);
void clearZ();
void clearY();
void clearX();
void write_horizon(struct orientationData * data);
void write_vertical_line(int x1, int y1, int y2);
void write_iron_sights();
int getOffset(int deg);
void drawPitchAndRoll(int* xline, int* yline, int offset, int sign);
void draw_right_vertical(int offset);
void draw_left_vertical(int offset);
void clear_right_vertical(int offset);
```

```
      void clear_left_vertical(int offset);
};

#endif /*LCD_H_*/
```

LCD Source

```
#include "LCD.h"
#include "io430x14x1.h"
#include "t6963c.h"
#include "in430.h"
#include "math.h"
#include "rateSensor.h"
#include <stdlib.h>
#define FIELD_LENGTH 6

/**
 * Break up sprites into x and y componets and apply to instance variables
for current and previous values
 */

int a0x [CROSSHAIR_LENGTH] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30};
int a0y [CROSSHAIR_LENGTH] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int a10x [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
int a10y [CROSSHAIR_LENGTH] = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5};
int a20x [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28};
int a20y [CROSSHAIR_LENGTH] = {0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5,
5, 5, 6, 6, 6, 7, 7, 7, 8, 8, 8, 9, 9, 9, 10};
int a30x [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11, 12,
12, 13, 14, 15, 16, 17, 18, 19, 19, 20, 21, 22, 23, 24, 25, 25};
int a30y [CROSSHAIR_LENGTH] = {0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7,
7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13, 14, 14};
int a40x [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 9, 10, 11,
12, 13, 13, 14, 15, 16, 16, 17, 18, 19, 19, 20, 21, 22, 22};
int a40y [CROSSHAIR_LENGTH] = {0, 1, 1, 2, 3, 3, 4, 5, 5, 6, 7, 7, 8, 8, 9,
10, 10, 11, 12, 12, 13, 14, 14, 15, 16, 16, 17, 17, 18, 19};
int a50x [CROSSHAIR_LENGTH] = {0, 1, 1, 2, 3, 3, 4, 5, 5, 6, 7, 7, 8, 8, 9,
10, 10, 11, 12, 12, 13, 14, 14, 15, 16, 16, 17, 17, 18, 19};
int a50y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 9, 10, 11,
12, 13, 13, 14, 15, 16, 16, 17, 18, 19, 19, 20, 21, 22, 22};
int a60x [CROSSHAIR_LENGTH] = {0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7,
8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13, 14, 14, 15};
int a60y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11, 12,
12, 13, 14, 15, 16, 17, 18, 19, 19, 20, 21, 22, 23, 24, 25, 25};
int a70x [CROSSHAIR_LENGTH] = {0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5,
5, 5, 6, 6, 6, 7, 7, 7, 8, 8, 8, 9, 9, 9, 10};
```

```
int a70y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28};
int a80x [CROSSHAIR_LENGTH] = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5};
int a80y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
int a90x [CROSSHAIR_LENGTH] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int a90y [CROSSHAIR_LENGTH] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30};
int a100x [CROSSHAIR_LENGTH] = {0, 0, 0, 0, 0, -1, -1, -1, -1, -1, -1, -2, -
2, -2, -2, -2, -2, -3, -3, -3, -3, -3, -3, -4, -4, -4, -4, -4, -5, -5};
int a100y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
int a110x [CROSSHAIR_LENGTH] = {0, 0, -1, -1, -1, -2, -2, -2, -3, -3, -3, -4,
-4, -4, -5, -5, -5, -6, -6, -6, -7, -7, -7, -8, -8, -8, -9, -9, -9, -10};
int a110y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28};
int a120x [CROSSHAIR_LENGTH] = {0, 0, -1, -1, -2, -2, -3, -3, -4, -4, -5, -5,
-6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14,
-14};
int a120y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11, 12,
12, 13, 14, 15, 16, 17, 18, 19, 19, 20, 21, 22, 23, 24, 25, 25};
int a130x [CROSSHAIR_LENGTH] = {0, -1, -1, -2, -3, -3, -4, -5, -5, -6, -7, -
7, -8, -8, -9, -10, -10, -11, -12, -12, -13, -14, -14, -15, -16, -16, -17, -
17, -18, -19};
int a130y [CROSSHAIR_LENGTH] = {0, 1, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 9, 10,
11, 12, 13, 13, 14, 15, 16, 16, 17, 18, 19, 19, 20, 21, 22, 22};
int a140x [CROSSHAIR_LENGTH] = {0, -1, -2, -3, -3, -4, -5, -6, -6, -7, -8, -
9, -9, -10, -11, -12, -13, -13, -14, -15, -16, -16, -17, -18, -19, -19, -20,
-21, -22, -22};
int a140y [CROSSHAIR_LENGTH] = {0, 1, 1, 2, 3, 3, 4, 5, 5, 6, 7, 7, 8, 8, 9,
10, 10, 11, 12, 12, 13, 14, 14, 15, 16, 16, 17, 17, 18, 19};
int a150x [CROSSHAIR_LENGTH] = {0, -1, -2, -3, -4, -5, -6, -6, -7, -8, -9, -
10, -11, -12, -12, -13, -14, -15, -16, -17, -18, -19, -19, -20, -21, -22, -
23, -24, -25, -25};
int a150y [CROSSHAIR_LENGTH] = {0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7,
7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13, 14, 14};
int a160x [CROSSHAIR_LENGTH] = {0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -
11, -12, -13, -14, -15, -15, -16, -17, -18, -19, -20, -21, -22, -23, -24, -
25, -26, -27, -28};
int a160y [CROSSHAIR_LENGTH] = {0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5,
5, 5, 6, 6, 6, 7, 7, 7, 8, 8, 8, 9, 9, 9, 10};
int a170x [CROSSHAIR_LENGTH] = {0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -
11, -12, -13, -14, -15, -16, -17, -18, -19, -20, -21, -22, -23, -24, -25, -
26, -27, -28, -29};
int a170y [CROSSHAIR_LENGTH] = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5};

LCD::LCD() {
}

int LCD::busy() {
      int status = 0;
      read_command();
      status = 0x03 & ~LCDIN;
      end();
```

```
	return status;
}

void LCD::read_command() {
	LCDDIR = 0;
	LCDCTRLOUT |= (LCDCOMMAND + LCDWRITE);
	LCDCTRLOUT &= ~LCDREAD;
	LCDCTRLOUT &= ~LCDENABLE;
}
void LCD::end() {
	LCDCTRLOUT |= LCDENABLE;
	LCDCTRLOUT |= (LCDCOMMAND+LCDREAD+LCDWRITE);
	LCDDIR=0;
}

void LCD::init() {
	LCDSEL = 0;
	LCDDIR = 0;
	LCDCTRLSEL &= ~LCDMASK;
	LCDCTRLDIR |= LCDMASK;
	LCDCTRLOUT |= LCDMASK;
	write_command(CGROM_EXOR_MODE);
	write_dataword(GRAPHICS_HOME_ADDRESS);
	write_command(GRAPHICS_HOME_ADDRESS_SET);
	write_dataword(GRAPHICS_AREA);
	write_command(GRAPHICS_AREA_SET);
	write_dataword(GRAPHICS_HOME_ADDRESS);
	write_dataword(TEXT_HOME_ADDRESS);
	write_command(TEXT_HOME_ADDRESS_SET);
	write_dataword(TEXT_AREA);
	write_command(TEXT_AREA_SET);
	//    write_command(DISPLAY_MODE+TEXT);
	write_command(DISPLAY_MODE+GRAPHICS+TEXT);
	clear_all_text();
	clear_display();
	//Draw data slots
	  write_text(8, 1, "G:");
	write_text(0, 0, "X:");
	write_text(8, 0, "Y:");
	  write_text(0, 1, "Z:");
	  write_text(8, 15, "R:");
	  write_text(0, 15, "P:");
	write_iron_sights();
	previousXLine = a0x;
	previousYLine = a0y;
	previousXSign = -1;
		previousYSign = 0;
	previousPitch = 0;
	previousRoll = 0;
}

void LCD::write_command(char a) {
	while (busy())
		;
	command_write();
	LCDOUT = a;
	end();
```

```
	return;
}

void LCD::command_write() {
	LCDCTRLOUT |= (LCDCOMMAND+LCDREAD);
	LCDCTRLOUT&=~LCDWRITE;
	LCDDIR|=0xFF;
	LCDCTRLOUT&=~LCDENABLE;
	return;
}
void LCD::data_write() {
	LCDCTRLOUT|=LCDREAD;
	LCDCTRLOUT&=~(LCDCOMMAND+LCDWRITE);
	LCDDIR|=0xFF;
	LCDCTRLOUT&=~LCDENABLE;
	return;
}
void LCD::data_read() {
	LCDDIR=0x00;
	LCDCTRLOUT|=LCDWRITE;
	LCDCTRLOUT&=~(LCDCOMMAND+LCDREAD);
	LCDCTRLOUT&=~LCDENABLE;
	return;
}

void LCD::write_dataword(int a) {
	write_data(a);
	write_data(_SWAP_BYTES(a));
	return;
}
void LCD::write_data(char a) {
	while (busy())
		;
	data_write();
	LCDOUT = a;
	end();
	return;
}

void LCD::write_text(int x, int y, char *g) {
	write_dataword(TEXT_HOME_ADDRESS + (y * TEXT_AREA) + x);
	write_command(ADDRESS_POINTER_SET);
	int z=0;
	if (g[0]==0x00)
		return;
	while (*g || z==0) {
		write_data((*g++) - 0x20);
		write_command(DATA_WRITE_UP);
		z++;
	}
	return;
}

void LCD::write_field(int x, int y, char *g) {
	write_dataword(TEXT_HOME_ADDRESS + (y * TEXT_AREA) + x);
	write_command(ADDRESS_POINTER_SET);
	int z=0;
```

```
	if (g[0]==0x00)
		return;
	while (*g && z<FIELD_LENGTH) {
		write_data((*g++) - 0x20);
		write_command(DATA_WRITE_UP);
		z++;
	}
	while (z < FIELD_LENGTH) {
		write_data(' ' - 0x20);
		write_command(DATA_WRITE_UP);
		z++;
	}
	return;
}

void LCD::clear_text(int x, int y, int w) {

	unsigned volatile int i;
	write_dataword(TEXT_HOME_ADDRESS + (y * TEXT_AREA) + x);
	write_command(ADDRESS_POINTER_SET);
	for (i=0; i<w; i++) {
		write_data(0);
		write_command(DATA_WRITE_UP);
	}
}

void LCD::clear_all_text() {
	unsigned volatile int s;
	int l = 18;
	for (s = 0; s < l; s++) {
		clear_text(0, s, 18);
	}
}

void LCD::clear_display(void) {

	unsigned volatile int i;
	write_dataword(GRAPHICS_HOME_ADDRESS);
	//	write_dataword(0);
	write_command(ADDRESS_POINTER_SET);
	for (i=0; i < 0x2000; i++) {//used to be 144
		write_data(0);
		write_command(DATA_WRITE_UP);
	}
}

void LCD::fill_display(void) {

	int i;
	//write_dataword(GRAPHICS_HOME_ADDRESS);
	write_dataword(0);
	write_command(ADDRESS_POINTER_SET);
	for (i=0; i<0x0028; i++) {
		write_data(1);
		write_command(DATA_WRITE_UP);
	}
}
```

```
void LCD::reset(void) {
	unsigned volatile int i;
	LCDCTRLDIR |= LCD_RESET;
	LCDCTRLOUT &= ~LCD_RESET;
	for (i=0; i<200; i++)
		;
	LCDCTRLOUT |= LCD_RESET;
}

void LCD::clear(int x, int y, int h, int w) {

	unsigned volatile int adres, i, j;
	adres=GRAPHICS_HOME_ADDRESS + (y * GRAPHICS_AREA) + x;
	for (i=0; i<h; i++) {
		write_dataword(adres);
		write_command(ADDRESS_POINTER_SET);
		for (j=0; j<w; j++) {
			write_data(0);
			write_command(DATA_WRITE_UP);
		}
		adres+=GRAPHICS_AREA;
	}
}

void LCD::write_horizontal_line(int x1, int y1, int x2) {
	int i;
	for (i = x1; i <= x2; i++) {
		lcd_set_pixel(i, y1);
	}
	return;
}

void LCD::write_vertical_line(int x1, int y1, int y2) {
	int i;
	for (i = y1; i <= y2; i++) {
		lcd_set_pixel(x1, i);
	}
	return;
}

void LCD::drawPitchAndRoll(int* xline, int* yline, int offset1, int xSign,
int ySign) {
	write_command(DISPLAY_MODE+GRAPHICS+TEXT);
	lcd_set_pixel(0, offset1);
	  for (int c = 0; c < CROSSHAIR_LENGTH; c++) {
		int xr_dot = CENTER_X + xline[c];
		int yr_dot = yline[c] + CENTER_Y + offset1;
		int xl_dot = CENTER_X - xline[c];
		int yl_dot = CENTER_Y - yline[c] + offset1;
		lcd_set_pixel(xr_dot, yr_dot);
		lcd_set_pixel(xl_dot, yl_dot);
	  }
	write_command(DISPLAY_MODE+GRAPHICS+TEXT);
}
```

```
/**
 * Change the write mode to be xor
 * Assess the line
 * Flip the line over the vertical axis
 * Store the line in an instance variable
 * Draw the line using the auto write up command
 * Be done
 */
void LCD::write_horizon(struct orientationData * data) {
      int roll = data->roll;
      int pitch = data-> pitch;
        roll = roll % 360;
        roll = abs(roll);
        int ySign, xSign;
        if(roll >= 180 && roll < 270) {
          roll = roll - 180;
          xSign = 1;
        } else if(roll >=90 && roll < 180) {
          xSign = -1;
        } else if(roll >=270) {
          roll = roll - 180;
          xSign = -1;
        } else {
          xSign = 1;
        }
      int *xLine, *yLine;
      int evenRoll = roll - (roll % 10);
      switch (evenRoll) {
            case 170:
                  xLine = a170x; yLine = a170y;
                  break;
            case 160:
                  xLine = a160x; yLine = a160y;
                  break;
            case 150:
                  xLine = a150x; yLine = a150y;
                  break;
            case 140:
                  xLine = a140x; yLine = a140y;
                  break;
            case 130:
                  xLine = a130x; yLine = a130y;
                  break;
            case 120:
                  xLine = a120x; yLine = a120y;
                  break;
            case 110:
                  xLine = a110x; yLine = a110y;
                  break;
            case 100:
                  xLine = a100x; yLine = a100y;
                  break;
            case 90:
                  xLine = a90x; yLine = a90y;
                  break;
            case 80:
                  xLine = a80x; yLine = a80y;
```

```
                break;
            case 70:
                xLine = a70x; yLine = a70y;
                break;
            case 60:
                xLine = a60x; yLine = a60y;
                break;
            case 50:
                xLine = a50x; yLine = a50y;
                break;
            case 40:
                xLine = a40x; yLine = a40y;
                break;
            case 30:
                xLine = a30x; yLine = a30y;
                break;
            case 20:
                xLine = a20x; yLine = a20y;
                break;
            case 10:
                xLine = a10x; yLine = a10y;
                break;
          default:
                xLine = a0x;
                yLine = a0y;
                break;
      }
      if (previousXSign != xSign || previousXLine != xLine) {
            drawPitchAndRoll(previousXLine, previousYLine, previousPitch,
                        previousXSign, previousYSign);
            previousXSign = xSign;
            previousPitch = pitch;
            previousRoll = roll;
            previousXLine = xLine;
            previousYLine = yLine;
            drawPitchAndRoll(previousXLine, previousYLine, previousPitch,
                        previousXSign, previousYSign);
      }
}

void LCD::write_iron_sights() {
      //    write_horizontal_line(4, 67, 122);
      //    write_vertical_line(63, 17, 122);
      //    //Draw tick marks
      //    //10 and -10
      //    write_horizontal_line(58, 47, 68);
      //    write_horizontal_line(58, 87, 68);
      //    //20 and -20
      //    write_horizontal_line(58, 27, 68);
      //    write_horizontal_line(58, 107, 68);
      return;
}

void LCD::setX(char *g) {
      write_field(2, 0, g);
}
```

```
void LCD::setY(char *g) {
	write_field(10, 0, g);
}

void LCD::setZ(char *g) {
	write_field(2, 1, g);
}
void LCD::setG(char *g) {
	write_field(10, 1, g);
}
void LCD::setRoll(char *g) {
	write_field(2, 15, g);
}
void LCD::setPitch(char *g) {
	write_field(10, 15, g);
}

unsigned char LCD::lcd_read_data(void) {
	unsigned char data;
	while (busy())
		;
	data_read();
	data = LCDIN;
	end();
	return data;
}

/*****************************************
 * Function name: lcd_set_pixel          *
 * Description:   Set a single Pixel on  *
 *                0 <= X <= LCD Width    *
 *                0 <= Y <= LCD Height   *
 *****************************************/
void LCD::lcd_set_pixel(unsigned char x, unsigned char y) {
	unsigned char data;
	unsigned int address;

	address = (y * GRAPHICS_AREA) + (x / 8) + GRAPHICS_HOME_ADDRESS;

	data = lcd_bit_to_byte(7 - (x % 8));

	write_data(address & 0xff);
	write_data(address >> 0x08);
	write_command(ADDRESS_POINTER_SET);

	/* Read existing display */
	write_command(LCD_DATA_READ_NO_INCREMENT);
	data ^= lcd_read_data();

	/* Write modified data */
	write_data(data);
	write_command(DATA_WRITE_UP);
	return;
}

void LCD::lcd_clear_pixel(unsigned char x, unsigned char y) {
	unsigned char data;
```

```
	unsigned int address;

	address = (y * GRAPHICS_AREA) + (x / 8) + GRAPHICS_HOME_ADDRESS;

	data = lcd_bit_to_byte(7 - (x % 8));

	write_data(address & 0xff);
	write_data(address >> 0x08);
	write_command(ADDRESS_POINTER_SET);

	/* Read existing display */
	write_command(LCD_DATA_READ_NO_INCREMENT);
	data = data ^ lcd_read_data();

	/* Write modified data */
	write_data(data);
	write_command(DATA_WRITE_UP);
	return;
}

unsigned char LCD::lcd_bit_to_byte(unsigned char bit) {
	switch (bit) {
	case 0:
		return 1;
	case 1:
		return 2;
	case 2:
		return 4;
	case 3:
		return 8;
	case 4:
		return 16;
	case 5:
		return 32;
	case 6:
		return 64;
	case 7:
		return 128;
	default:
		return 0;
	}
}

void LCD::draw_right_vertical(int offset1) {
	lcd_line(60, 22 + offset1, 66, 112 + offset1);
	return;
}

void LCD::draw_left_vertical(int offset1) {
	lcd_line(60, 112 + offset1, 66, 22 + offset1);
	return;
}

void LCD::clear_right_vertical(int offset1) {
	clear_line(60, 22 + offset1, 66, 112 + offset1);
	return;
}
```

```
void LCD::clear_left_vertical(int offset1) {
	clear_line(66, 22 + offset1, 60, 112 + offset1);
	return;
}

//void LCD::clearPitchAndRoll(int offset1, int angle) {
//	if (angle > 45) {
//		clear_right_vertical(offset1);
//	}
//	//  else if(angle > 40) {
//	//		clear_right_four_five(offset1);
//	//  } else if(angle > 25) {
//	//		clear_right_three_zero(offset1);
//	//  } else if (angle > 10) {
//	//		clear_right_one_five(offset1);
//	//  } else if (angle >= -10) {
//	//		clear_zero(offset1);
//	//  } else if(angle > -25) {
//	//		clear_left_one_five(offset1);
//	//  } else if(angle > -40) {
//	//		clear_left_three_zero(offset1);
//	//  } else if(angle > -50) {
//	//		clear_left_four_five(offset1);
//	//  }
//	else
//		clear_left_vertical(offset1);
//	return;
//}

int LCD::getOffset(int deg) {
	if (deg >= 15) {
		return 20;
	} else if (deg >= 5) {
		return 10;
	} else if (deg >= 0) {
		return 0;
	} else if (deg >= -5) {
		return -10;
	} else
		return -20;
}

int LCD::clear_line(int X1, int Y1, int X2, int Y2) {
	int t, distance;
	int xerr, yerr;
	int Dx, Dy;
	int incx, incy;
	if (X2 > 127) {
		X2 = 127;
	} else if (X2 < 1) {
		X2 = 1;
	}
	if (Y2 > 127) {
		Y2 = 127;
	} else if (Y2 < 17) {
		Y2 = 17;
```

```
}
Dx = X2 - X1;
Dy = Y2 - Y1;
if (Dx > 0) {
      incx = 1;
} else {
      if (Dx == 0)
            incx = 0;
      else
            incx = -1;
}
if (Dy > 0) {
      incy = 1;
} else {
      if (Dy == 0)
            incy = 0;
      else
            incy = -1;
}
if (Dy < 0) {
      Dy = -Dy;
}
if (Dx < 0) {
      Dx = -Dx;
}

if (Dx > Dy) {
      distance = Dx;
} else {
      distance = Dy;
}

yerr = 0;
xerr = 0;
for (t = 0; t <= distance + 1; t++) {
      lcd_clear_pixel(X1, Y1);
      xerr += Dx;
      yerr += Dy;
      if (xerr > distance) {
            xerr -= distance;
            X1 += incx;
            if (X1 < 0) {
                  X1 = 0;
            }
            if (X1 > X2) {
                  X1 = X2;
            }
      }
      if (yerr > distance) {
            yerr -= distance;
            Y1 += incy;
            if (Y1 < 0) {
                  Y1 = 0;
            }
            if (Y1 > Y2) {
                  Y1 = Y2;
            }
```

```
            }
        }
        write_iron_sights();
        return 0;

}
```

T6963 (LCD Processor) header

```
#include "io430x14x1.h"
#ifndef __t6963c
#define __t6963c

typedef char graphics[];// {char width, char height, char pattern.....}
typedef struct
{
    float x, y;
    float xe, ye;
    float heading, magnitude;
    float temperature;
    unsigned int distortion, calStatus;
} V2XEData;

void lcd_init (void);
void lcd_clear_display(void);

//x, y, height(pixels), width(columns);
void lcd_write_graphics(int x, int y, const graphics g);
void lcd_clear(int x, int y, int h, int w);
void lcd_write_text(int x, int y, const char *g);//g[] must end with 0x00
void lcd_clear_text(int x, int y, int w);
#define LCDIN   P5IN
#define LCDOUT  P5OUT   //databus to/from LCD
#define LCDDIR  P5DIR
#define LCDSEL  P5SEL

#define LCDCTRLIN   P4IN    //e.g.  P5
#define LCDCTRLOUT  P4OUT
#define LCDCTRLDIR  P4DIR
#define LCDCTRLSEL  P4SEL
#define LCDCOMMAND  (0x01)  //P6.0
#define LCDREAD     (0x02)  //P6.1  control pins in LCDCTRLxxx port
#define LCDWRITE    (0x04)  //P6.2
#define LCDENABLE   (0x08)  //P6.3

#define TEXT_HOME_ADDRESS (0x1400)
#define TEXT_AREA (0x0012)
#define GRAPHICS_HOME_ADDRESS (0x0000) //Set the graphics home address four
lines down from the top of screen
```

```
#define GRAPHICS_AREA (0x0012)

void lcd_write_data(char a);
void lcd_write_dataword(int a);
void lcd_write_command(char a);

#define CURSOR_POINTER_SET (0x21)
#define OFFSET_REGISTER_SET (0x22)
#define ADDRESS_POINTER_SET (0x24)
#define TEXT_HOME_ADDRESS_SET (0x40)
#define TEXT_AREA_SET (0x41)
#define GRAPHICS_HOME_ADDRESS_SET (0x42)
#define GRAPHICS_AREA_SET (0x43)
#define CGROM_OR_MODE   (0x80)
#define CGROM_EXOR_MODE (0x81)
#define CGROM_AND_MODE  (0x83)
#define CGROM_TEXT_MODE (0x84)
#define CGRAM_OR_MODE   (0x88)
#define CGRAM_EXOR_MODE (0x89)
#define CGRAM_AND_MODE  (0x8B)
#define CGRAM_TEXT_MODE (0x8C)

#define DISPLAY_MODE  (0x90)
#define GRAPHICS (0x08)
#define TEXT     (0x04)
#define CURSOR   (0x02)
#define BLINK    (0x01)

//Assign variables to Bit Set / Reset

#define CURSOR_BOTTOM (0xA0)
#define CURSOR_2LINE  (0xA1)
#define CURSOR_3LINE  (0xA2)
#define CURSOR_4LINE  (0xA3)
#define CURSOR_5LINE  (0xA4)
#define CURSOR_6LINE  (0xA5)
#define CURSOR_7LINE  (0xA6)
#define CURSOR_8LINE  (0xA7)

#define DATA_AUTO_WRITE (0xB0)
#define DATA_AUTO_READ  (0xB1)
#define AUTO_RESET      (0xB2)
#define LCD_RESET       (0x10)
#define LCD_DATA_WRITE_NO_INCREMENT     (0xC4)
#define LCD_DATA_READ_NO_INCREMENT      (0xC5)

#define DATA_READ_UP    (0xC1)
#define DATA_READ_DOWN  (0xC3)
#define DATA_WRITE_UP   (0xC0)
#define DATA_WRITE_DOWN (0xC2)
#define LCDMASK   (LCDCOMMAND + LCDREAD + LCDWRITE + LCDENABLE)
#endif
```

T6963 (LCD Processor) Source

```
#include "t6963c.h"

#define LCDMASK     (LCDCOMMAND+LCDREAD+LCDWRITE+LCDENABLE)

//////////////
#define NEG_CTRL //ze wzgledu na polaczenia elektryczne
                 //i zmiane poziomow (3.3V <=> 5V)
                 //wymagana jest odwrotna polaryzacja sygnalow sterujacych
#ifdef  NEG_CTRL

#define LCD_COMMAND_WRITE() LCDCTRLOUT&=~(LCDCOMMAND+LCDREAD); \
                            LCDCTRLOUT|=LCDWRITE; \
                            LCDDIR=-1; \
                            LCDCTRLOUT|=LCDENABLE;
#define LCD_COMMAND_READ()  LCDDIR=0; \
                            LCDCTRLOUT&=~(LCDCOMMAND+LCDWRITE); \
                            LCDCTRLOUT|=LCDREAD; \
                            LCDCTRLOUT|=LCDENABLE;
#define LCD_DATA_WRITE()    LCDCTRLOUT&=~LCDREAD; \
                            LCDCTRLOUT|=(LCDCOMMAND+LCDWRITE); \
                            LCDDIR=-1; \
                            LCDCTRLOUT|=LCDENABLE;
#define LCD_DATA_READ()     LCDDIR=0; \
                            LCDCTRLOUT&=~LCDWRITE; \
                            LCDCTRLOUT|=(LCDCOMMAND+LCDREAD); \
                            LCDCTRLOUT|=LCDENABLE;
#define LCD_END()           LCDCTRLOUT&=~LCDENABLE; \
                            LCDCTRLOUT&=~(LCDCOMMAND+LCDREAD+LCDWRITE); \
                            LCDDIR=0;
#else

#define LCD_COMMAND_WRITE() LCDCTRLOUT|=(LCDCOMMAND+LCDREAD); \
                            LCDCTRLOUT&=~LCDWRITE; \
                            LCDDIR=-1; \
                            LCDCTRLOUT&=~LCDENABLE;
#define read_command()  LCDDIR=0; \
                            LCDCTRLOUT|=(LCDCOMMAND+LCDWRITE); \
                            LCDCTRLOUT&=~LCDREAD; \
                            LCDCTRLOUT&=~LCDENABLE;
#define LCD_DATA_WRITE()    LCDCTRLOUT|=LCDREAD; \
                            LCDCTRLOUT&=~(LCDCOMMAND+LCDWRITE); \
                            LCDDIR=-1; \
                            LCDCTRLOUT&=~LCDENABLE;
#define LCD_DATA_READ()     LCDDIR=0; \
                            LCDCTRLOUT|=LCDWRITE; \
                            LCDCTRLOUT&=~(LCDCOMMAND+LCDREAD); \
                            LCDCTRLOUT&=~LCDENABLE;
#define end()           LCDCTRLOUT|=LCDENABLE; \
                            LCDCTRLOUT|=(LCDCOMMAND+LCDREAD+LCDWRITE); \
                            LCDDIR=0;
#endif
/////////
```

```
int lcd_busy (void);
int lcd_busy_a_w (void); //busy auto write

void lcd_init (void){
  LCDSEL = 0;//piny portu komuniakcji z wyœwietlaczem jako I/O
  LCDDIR = 0;//piny portu ustawione jako wejscia ( w razie czego)
  LCDCTRLSEL   &=  ~LCDMASK;//ustawienie  odpowiednich  pinow  sterujacych  jako
I/O
  LCDCTRLDIR  |= LCDMASK;//piny kontrolne jako wyjscia
#ifdef  NEG_CTRL
  LCDCTRLOUT &= ~LCDMASK;//ustawienie 'jedynek' na pinach steruj¹cych LCD
#else
  LCDCTRLOUT |= LCDMASK;//ustawienie 'jedynek' na pinach steruj¹cych LCD
#endif

//inicjacja dla LCD  240x128 (40 kolumn => matryca znakow 6x8)
  lcd_write_command(CGROM_OR_MODE);
  lcd_write_dataword(GRAPHICS_HOME_ADDRESS);
  lcd_write_command(GRAPHICS_HOME_ADDRESS_SET);
  lcd_write_dataword(GRAPHICS_AREA);
  lcd_write_command(GRAPHICS_AREA_SET);
  lcd_write_dataword(GRAPHICS_HOME_ADDRESS);
  lcd_write_dataword(TEXT_HOME_ADDRESS);
  lcd_write_command(TEXT_HOME_ADDRESS_SET);
  lcd_write_dataword(TEXT_AREA);
  lcd_write_command(TEXT_AREA_SET);
  lcd_write_command(DISPLAY_MODE+GRAPHICS+TEXT);

  lcd_clear_display();
}


int lcd_busy (void){

  int status;
  LCD_COMMAND_READ();
  status = 0b00000011 & ~LCDIN;
  LCD_END();
  return status;//status=0 - ready; status>0 - busy
//  return 0; //do debagowania tylko!!!!!
}

int lcd_busy_a_w (void){

  int status;
  LCD_COMMAND_READ();
  status = 0b00001000 & ~LCDIN;
  LCD_END();
  return status;//status=0 - ready; status>0 - busy
//  return 0; //do debagowania tylko!!!!!
}

void lcd_write_data(char a){

  while(lcd_busy());
  LCD_DATA_WRITE();
```

```
  LCDOUT = a;
  LCD_END();
}

void lcd_write_data_a(char a){

  while(lcd_busy_a_w());
  LCD_DATA_WRITE();
  LCDOUT = a;
  LCD_END();
}

void lcd_write_dataword(int a){

  lcd_write_data(a);//bierze tylko mlodszy bajt 'a'
  write_data(_SWPB(a));//zamiana aby wzi¹æ starszy bajt 'a'
}

void lcd_write_command(char a){

  while(lcd_busy());
  LCD_COMMAND_WRITE();
  LCDOUT = a;
  LCD_END();
}

void lcd_write_command_a(char a){

  while(lcd_busy_a_w());
  LCD_COMMAND_WRITE();
  LCDOUT = a;
  LCD_END();
}

void lcd_clear_display(void){

  int i;
//  write_dataword(GRAPHICS_HOME_ADDRESS);
  lcd_write_dataword(0);//lepiej tak bo i tak jest czyszczonya ca³a pamiêæ
  lcd_write_command(ADDRESS_POINTER_SET);
  for(i=0;i<0x2000;i++){//ca³a pamiêc RAM zostanie wyczyszczona (wstawione
'0')
    lcd_write_data(0);
    lcd_write_command(DATA_WRITE_UP);
  }
}

void lcd_clear_display_a(void){

  int i;
//  write_dataword(GRAPHICS_HOME_ADDRESS);
  lcd_write_dataword(0);//lepiej tak bo i tak jest czyszczonya ca³a pamiêæ
  lcd_write_command(ADDRESS_POINTER_SET);
  lcd_write_command(DATA_AUTO_WRITE);
  while(lcd_busy());
  for(i=0;i<0x2000;i++) lcd_write_data_a(0);
  lcd_write_command_a(AUTO_RESET);
```

```
}

//x, y, height(pixels), width(columns)
// x,y=0,0 =>left top corner
void lcd_clear(int x, int y, int h, int w){

  int adres, i, j;
  adres=GRAPHICS_HOME_ADDRESS + (y * GRAPHICS_AREA) + x;
  for(i=0; i<h; i++){
    lcd_write_dataword(adres);
    lcd_write_command(ADDRESS_POINTER_SET);
    for(j=0; j<w; j++){
      lcd_write_data(0);
      lcd_write_command(DATA_WRITE_UP);
    }
    adres+=GRAPHICS_AREA;
  }
}

void lcd_clear_a(int x, int y, int h, int w){

  int adres, i, j;
  adres=GRAPHICS_HOME_ADDRESS + (y * GRAPHICS_AREA) + x;
  for(i=0; i<h; i++){
    lcd_write_dataword(adres);
    lcd_write_command(ADDRESS_POINTER_SET);
    lcd_write_command(DATA_AUTO_WRITE);
    while(lcd_busy());
    for(j=0; j<w; j++) lcd_write_data_a(0);
    lcd_write_command_a(AUTO_RESET);
    adres+=GRAPHICS_AREA;
  }
}

//[x,y, height(pixels), width(columns); x,y=0,0 =>left top corner; p[]=char
pattern
void lcd_write_graphics_a(int x, int y, const graphics g){

  int adres, i, j;
  int w=(char)(*g++);//width
  int h=(char)(*g++);//height

  adres=GRAPHICS_HOME_ADDRESS + (y * GRAPHICS_AREA) + x;
  for(i=0; i<h; i++){
    lcd_write_dataword(adres);
    lcd_write_command(ADDRESS_POINTER_SET);
    lcd_write_command(DATA_AUTO_WRITE);
    while(lcd_busy());
    for(j=0; j<w; j++)lcd_write_data_a(*g++);
    lcd_write_command_a(AUTO_RESET);
    adres+=GRAPHICS_AREA;
  }
}

//[x,y, height(pixels), width(columns); x,y=0,0 =>left top corner; p[]=char
pattern
void lcd_write_graphics(int x, int y, const graphics g){
```

```
  int adres, i, j;
  int w=(char)(*g++);//width
  int h=(char)(*g++);//height

  adres=GRAPHICS_HOME_ADDRESS + (y * GRAPHICS_AREA) + x;
  for(i=0; i<h; i++){
    lcd_write_dataword(adres);
    lcd_write_command(ADDRESS_POINTER_SET);
    for(j=0; j<w; j++){
      write_data(*g++);
      lcd_write_command(DATA_WRITE_UP);
    }
    adres+=GRAPHICS_AREA;
  }
}

void lcd_write_text(int x, int y, const char *g){

  lcd_write_dataword(TEXT_HOME_ADDRESS + (y * TEXT_AREA) + x);
  lcd_write_command(ADDRESS_POINTER_SET);
  while(*g){
    lcd_write_data((*g++) - 0x20);
    lcd_write_command(DATA_WRITE_UP);
  }
}

void lcd_write_text_a(int x, int y, const char *g){

  lcd_write_dataword(TEXT_HOME_ADDRESS + (y * TEXT_AREA) + x);
  lcd_write_command(ADDRESS_POINTER_SET);
  lcd_write_command(DATA_AUTO_WRITE);
  while(lcd_busy());
  while(*g)lcd_write_data_a((*g++) - 0x20);
  lcd_write_command_a(AUTO_RESET);
}

void lcd_clear_text(int x, int y, int w){

  int i;
  lcd_write_dataword(TEXT_HOME_ADDRESS + (y * TEXT_AREA) + x);
  lcd_write_command(ADDRESS_POINTER_SET);
  for(i=0; i<w; i++){
    lcd_write_data(0);
    lcd_write_command(DATA_WRITE_UP);
  }
}

void lcd_clear_text_a(int x, int y, int w){

  int i;
  lcd_write_dataword(TEXT_HOME_ADDRESS + (y * TEXT_AREA) + x);
  lcd_write_command(ADDRESS_POINTER_SET);
  lcd_write_command(DATA_AUTO_WRITE);
  while(lcd_busy());
  for(i=0; i<w; i++)lcd_write_data_a(0);
  lcd_write_command_a(AUTO_RESET);
```

```
}
```

Sonic Alert Header

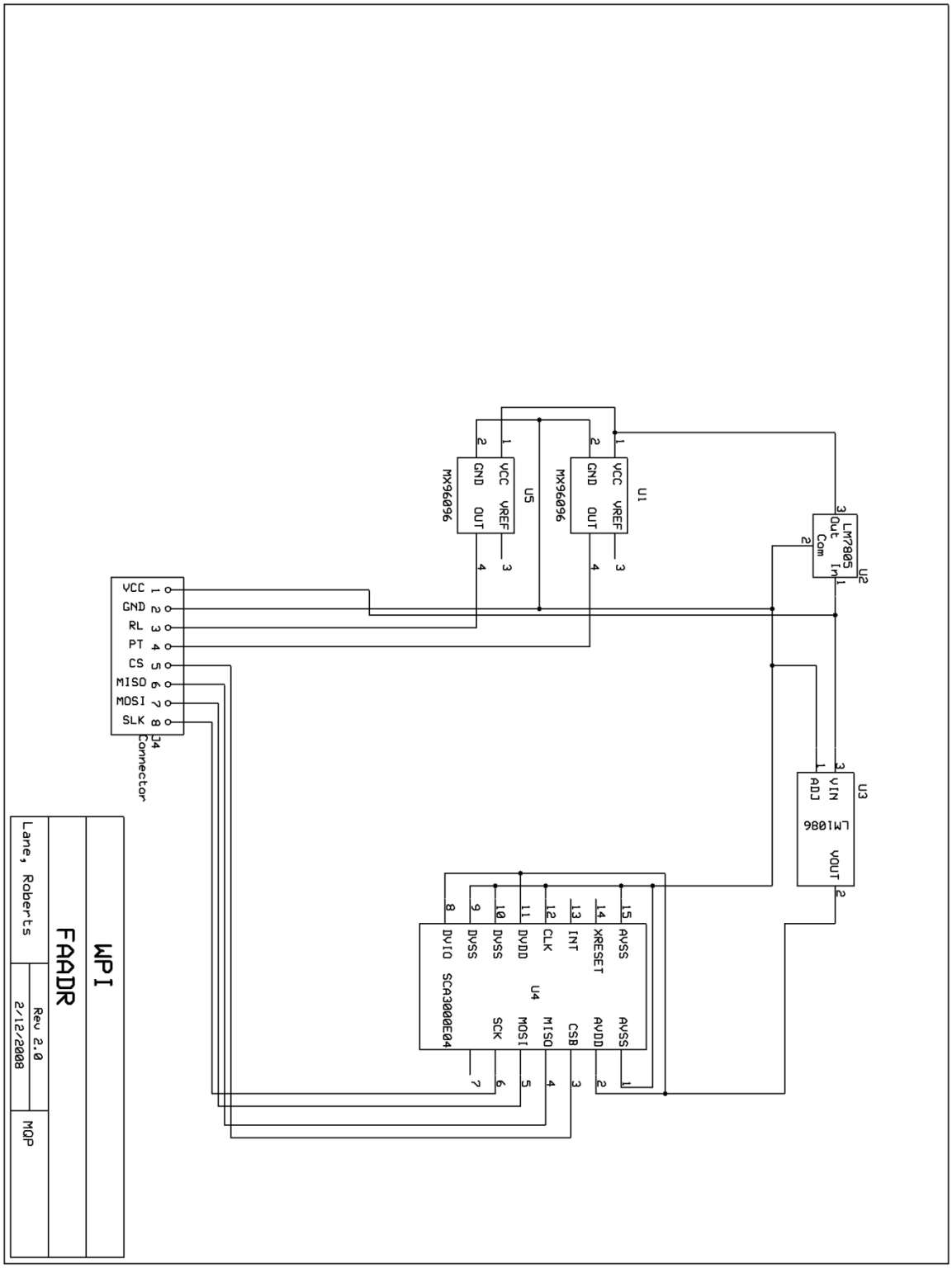
```
#ifndef SONICALERT_H_
#define SONICALERT_H_

class sonicAlert
{
public:
      sonicAlert();
      void start();
      void stop();
};

#endif /*SONICALERT_H_*/
```

Sonic Alert Source

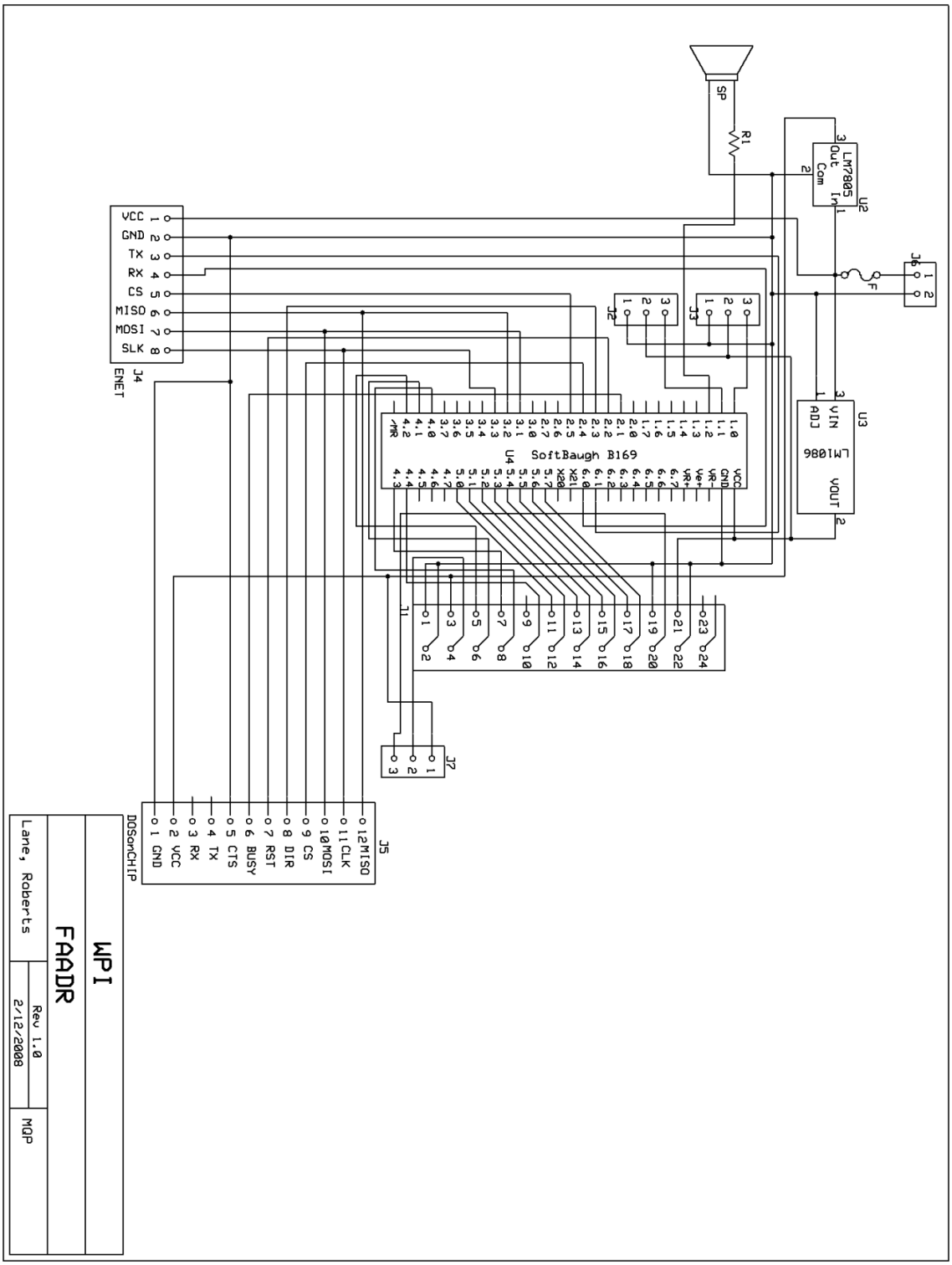
```
#include "sonicAlert.h"
#include "msp430x14x1.h"

sonicAlert::sonicAlert()
{
//    WDTCTL = WDTPW + WDTHOLD;     // Stop watchdog timer
      P1DIR |= BIT0;                      // Set P1.0 to output direction
      stop();
}

void sonicAlert::start(){
      P1OUT |= BIT0;    //Set pin 2.1 to output high
}

void sonicAlert::stop(){
      P1OUT &= ~BIT0;   //Set pin 2.1 to output low
}
```

Appendix B – Design Schematics

Sensor Board

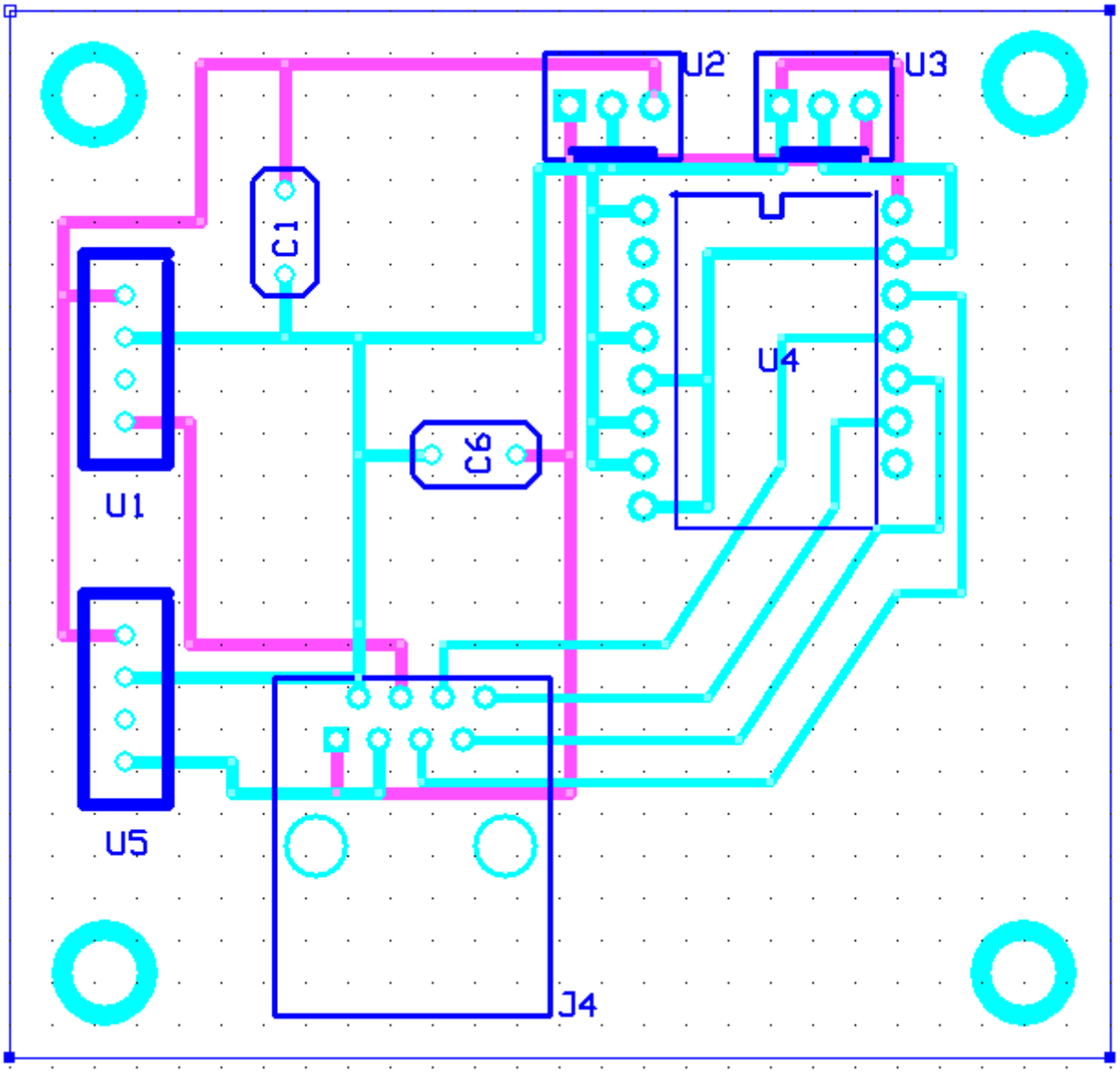


Main Board

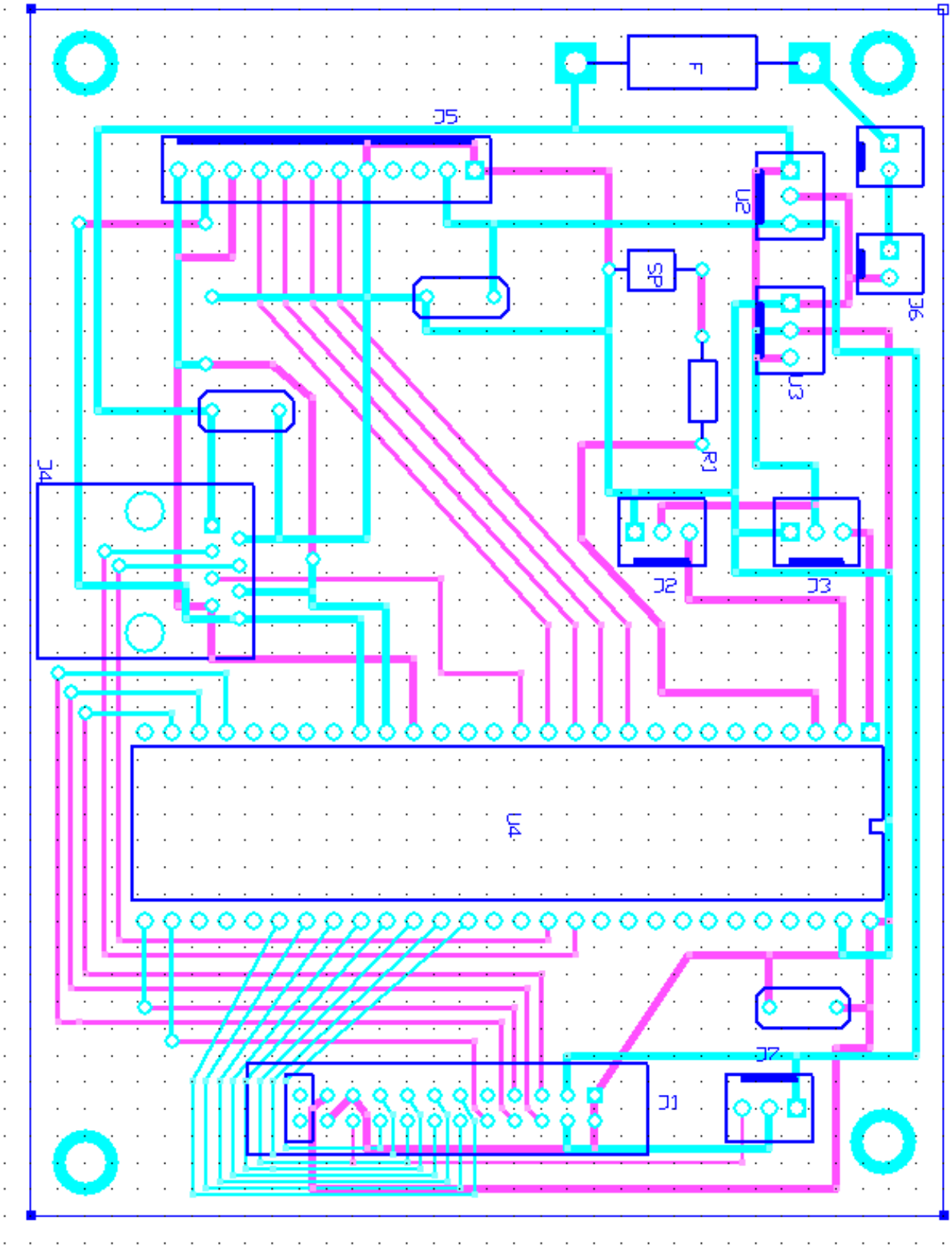

SP
R1
LM7805
Out
Com
In
U2
VCC
GND
TX
RX
CS
MISO
MOSI
SLK
J4
ENET
J2
J3
J6
F
SoftBaugh B169
U4
LM1086
ADJ
VIN
VOUT
U3
J1
J7
J5
DOSonCHIP
GND
VCC
RX
TX
CTS
BUSY
RST
DIR
CS
MOSI
CLK
MISO
WPI
FAADR
Lane, Roberts
Rev 1.0
2/12/2008
MQP

Appendix C – PCB Board Design

Sensor Board



Main Board



Appendix D – Horizon Sprite and Logic Generator Source Code

```
import java.util.ArrayList;
import java.util.Arrays;

public class HorizonSprite {

	private static final String CASE_VARIABLE = "evenRoll";
	public static Double offset = 0.0;
	public static Integer[] angles = { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90,
			100, 110, 120, 130, 140, 150, 160, 170 };
	public static int length = 30;
	public static int[][] trace = { { 1, 1, 1 }, { 2, 2, 2 } };
	public static String variable = "roll";
	public static String xPointer = "xLine";
	public static String yPointer = "yLine";

	public static void main(String args[]) {
		Arrays.sort(angles);
		for (Integer angle : angles) {
			ArrayList<String> xCoordinate = new ArrayList<String>();
			ArrayList<String> yCoordinate = new ArrayList<String>();
			Double newAngle = (angle * (Math.PI / 180));
			for (int i = 1; i <= length; i++) {
				xCoordinate.add((Integer
						.toString((int) (Math.cos(newAngle) * i))));
				yCoordinate.add((Integer
						.toString((int) (Math.sin(newAngle) * i))));
			}
			;
			String xCoordString = xCoordinate.toString().replace("[", "{");
			xCoordString = xCoordString.replace("]", "};");
			String yCoordString = yCoordinate.toString().replace("[", "{");
			yCoordString = yCoordString.replace("]", "};");
			System.out.println("int a" + angle + "x [CROSSHAIR_LENGTH] = "
					+ xCoordString);
			System.out.println("int a" + angle + "y [CROSSHAIR_LENGTH] = "
					+ yCoordString);
		}
		System.out.println("\n\n");
		Integer[] angles = HorizonSprite.angles;
		System.out.println("switch (" + CASE_VARIABLE + ") {");
		Arrays.sort(angles);
		for (int i = angles.length - 1; i > 0; i--) {
			processCase(angles[i]);
		}
		System.out.println("}");
	}

	public static void processCase(Integer angle) {
		System.out.println("\tcase " + angle + ":");
		System.out.println("\t\t" + xPointer + " = a" + angle + "x; "
				+ yPointer + " = a" + angle + "y;");
		System.out.println("\t\tbreak;");
	}
}
```

Appendix E - Analysis Software Source Code

Main Analysis Class

```
package gui;

import java.awt.Image;
import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.TreeSet;

import javax.imageio.ImageIO;
import javax.swing.Icon;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import thinlet.FrameLauncher;
import thinlet.Thinlet;

public class Starter extends Thinlet {

	private static final long serialVersionUID = 0;
	private static String label1 = "label1";
	private static String label2 = "label2";
	private static String label3 = "label3";
	private static String label4 = "label4";
	private static String label5 = "label5";
	private static String label6 = "label6";
	private static String label7 = "label7";
	private static String label8 = "label8";
	private static String label9 = "label9";
	private static String label10 = "label10";
	private static String dir = "dir";
	private static String dirButton = "dirButton";
	private static String fileList = "fileList";
	private static String path = "path";
	private static String pathButton = "pathButton";
	private static FrameLauncher frame;
	private static String lineText = "Total: Line Chart";
	private static String barText = "Total: Bar Chart";
	private static String pLineText = "Pitch: Line Chart";
	private static String rLineText = "Roll: Line Chart";
	private static String yLineText = "Yaw: Line Chart";
```

```
	private static String detailsText = "Details";
	private static String chart = "chart";
	private String directory = null;
	private static final String lineChart = "lineChart.png";
	private static final String pLineChart = "pLineChart.png";
	private static final String rLineChart = "rLineChart.png";
	private static final String yLineChart = "yLineChart.png";
	private static final String barChart = "barChart.png";


	public Starter(int screen) throws Exception {
		if(screen==1)
			add(parse("gui.xml"));
		else if(screen==2)
			add(parse("wait.xml"));
		else if(screen==3)
			add(parse("gui2.xml"));
	}

	public static void main(String[] args) throws Exception {

		frame = new FrameLauncher("FAADR Analysis Start Page",
					new Starter(1), 320, 240);
	}

	public void getFiles(String dir) {
		Object fileList = find("fileList");
		StringBuffer sb = new StringBuffer();
		File directory = new File(dir);
		String[] children = directory.list();
	    if (children == null) {
		setString(fileList, "text", "");
	    } else {
		  for (int i=0; i<children.length; i++) {
		      if(children[i].contains("txt"))
				sb.append(children[i]+"\n");
		  }
		  setString(fileList, "text", sb.toString());
	    }
	}


	private void removePathSelect(){
		makeCharts();
		frame.dispose();
		try {
			frame = new FrameLauncher("FAADR Chart Analysis", new
Starter(3), 330, 350);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}


	private void makeLineChart(TreeSet<Data>data){
		final XYDataset dataset = DataSets.createTotalDataSet(data);
		final JFreeChart chart = ChartFactory.createXYLineChart(
```

```
            "Total Acceleration",     // chart title
            "Time",                   // x axis label
            "Total Acceleration",     // y axis label
            dataset,                  // data
            PlotOrientation.VERTICAL,
            false,                     // include legend
            true,                     // tooltips
            false                     // urls
        );
      final ChartPanel chartPanel = new ChartPanel(chart);
     chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
     try {
       ChartUtilities.saveChartAsPNG(new File(lineChart), chart,
                   getWidth(), getHeight());
      } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
      }
}

private void makeBarChart(TreeSet<Data>data){
      final CategoryDataset dataset = DataSets.createBarDataset(data);
      final JFreeChart chart = ChartFactory.createBarChart(
                  "Acceleration Over Rating",     // chart title
            "Time",                    // x axis label
            "Times Over Rating",      // y axis label
            dataset,                  // data
            PlotOrientation.VERTICAL,
            false,                     // include legend
            false,                     // tooltips
            false                     // urls
        );
      final ChartPanel chartPanel = new ChartPanel(chart);
     chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
     try {
       ChartUtilities.saveChartAsPNG(new File(barChart), chart,
                  getWidth(), getHeight());
      } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
      }
}

private void makePLineChart(TreeSet<Data>data){
      final XYDataset dataset = DataSets.createPDataSet(data);
      final JFreeChart chart = ChartFactory.createXYLineChart(
                  "Pitch Acceleration",     // chart title
              "Time",                   // x axis label
              "Pitch Acceleration",     // y axis label
            dataset,                  // data
            PlotOrientation.VERTICAL,
            false,                     // include legend
            true,                     // tooltips
            false                     // urls
        );
      final ChartPanel chartPanel = new ChartPanel(chart);
     chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
```

```
      try {
        ChartUtilities.saveChartAsPNG(new File(pLineChart), chart,
                    getWidth(), getHeight());
       } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
       }
}

private void makeRLineChart(TreeSet<Data>data){
       final XYDataset dataset = DataSets.createRDataSet(data);;
       final JFreeChart chart = ChartFactory.createXYLineChart(
            "Roll Acceleration",      // chart title
            "Time",                    // x axis label
            "Roll Acceleration",      // y axis label
            dataset,                   // data
            PlotOrientation.VERTICAL,
            false,                      // include legend
            true,                     // tooltips
            false                     // urls
          );
       final ChartPanel chartPanel = new ChartPanel(chart);
      chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
      try {
        ChartUtilities.saveChartAsPNG(new File(rLineChart), chart,
                    getWidth(), getHeight());
       } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
       }
}

private void makeYLineChart(TreeSet<Data>data){
       final XYDataset dataset = DataSets.createYDataSet(data);;
       final JFreeChart chart = ChartFactory.createXYLineChart(
            "Yaw Acceleration",      // chart title
            "Time",                    // x axis label
            "Yaw Acceleration",      // y axis label
            dataset,                   // data
            PlotOrientation.VERTICAL,
            false,                      // include legend
            true,                     // tooltips
            false                     // urls
          );
       final ChartPanel chartPanel = new ChartPanel(chart);
      chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
      try {
        ChartUtilities.saveChartAsPNG(new File(yLineChart), chart,
                    getWidth(), getHeight());
       } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
       }
}

private void makeCharts(){
       TreeSet<Data> data = DataParser.parseFile(directory);
```

```
            makeLineChart(data);
            makeBarChart(data);
            makePLineChart(data);
            makeRLineChart(data);
            makeYLineChart(data);
      }

      public void getData(String dir) {
            directory = dir;
            removePathSelect();
      }

      public void displayChart(String type) throws IOException{

            Object thisChart = find(chart);
            if(type.equals(lineText)){
                  Image img = ImageIO.read(new File(lineChart));
                  setIcon(thisChart, "icon", img);
            }
            else if(type.equals(barText)){
                  Image img = ImageIO.read(new File(barChart));
                  setIcon(thisChart, "icon", img);
            }
            else if(type.equals(rLineText)){
                  Image img = ImageIO.read(new File(rLineChart));
                  setIcon(thisChart, "icon", img);
            }
            else if(type.equals(pLineText)){
                  Image img = ImageIO.read(new File(pLineChart));
                  setIcon(thisChart, "icon", img);
            }
            else if(type.equals(yLineText)){
                  Image img = ImageIO.read(new File(yLineChart));
                  setIcon(thisChart, "icon", img);
            }
            else{
                  frame.dispose();
                  System.exit(-1);
            }

      }
}
```

Data Set Handler Class

```
package gui;

import java.util.Iterator;
import java.util.TreeSet;

import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
```

```
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

public class DataSets {

	public static XYDataset createYDataSet(TreeSet <Data> data){
	  final XYSeries series1 = new XYSeries("First");
	  for (Iterator<Data> iterator = data.iterator(); iterator.hasNext();)
{
			Data dataPoint = iterator.next();

	series1.add(dataPoint.getTime(),dataPoint.getYawAcceleration());
		}
	  final XYSeriesCollection dataset = new XYSeriesCollection();
	  dataset.addSeries(series1);
	  return dataset;
	}

	public static XYDataset createRDataSet(TreeSet <Data> data){
	  final XYSeries series1 = new XYSeries("First");
	  for (Iterator<Data> iterator = data.iterator(); iterator.hasNext();)
{
			Data dataPoint = iterator.next();

	series1.add(dataPoint.getTime(),dataPoint.getRollAcceleration());
		}
	  final XYSeriesCollection dataset = new XYSeriesCollection();
	  dataset.addSeries(series1);
	  return dataset;
	}

	public static XYDataset createPDataSet(TreeSet <Data> data){
	  final XYSeries series1 = new XYSeries("First");
	  for (Iterator<Data> iterator = data.iterator(); iterator.hasNext();)
{
			Data dataPoint = iterator.next();

	series1.add(dataPoint.getTime(),dataPoint.getPitchAcceleration());
		}
	  final XYSeriesCollection dataset = new XYSeriesCollection();
	  dataset.addSeries(series1);
	  return dataset;
	}

	public static XYDataset createTotalDataSet(TreeSet <Data> data){
	  final XYSeries series1 = new XYSeries("First");
	  for (Iterator<Data> iterator = data.iterator(); iterator.hasNext();)
{
			Data dataPoint = iterator.next();
			Double total = dataPoint.getYawAcceleration()+
				dataPoint.getPitchAcceleration()+
					dataPoint.getRollAcceleration();
			series1.add(dataPoint.getTime(),total);
		}
	  final XYSeriesCollection dataset = new XYSeriesCollection();
	  dataset.addSeries(series1);
```

```
        return dataset;
    }

    public static CategoryDataset createBarDataset(TreeSet <Data> data) {

                final String series1 = "Over Load";

            final String category1 = "Total";
            final String category2 = "Pitch";
            final String category3 = "Roll";
            final String category4 = "Yaw";

            // create the dataset...
            final DefaultCategoryDataset dataset = new
DefaultCategoryDataset();
            int pitch = 0,roll = 0,yaw = 0, total = 0;
            for (Iterator<Data> iterator = data.iterator();
iterator.hasNext();) {
                    Data dataPoint = iterator.next();
                    if(dataPoint.getPitchAcceleration()>5){
                          pitch+=1;
                          total+=1;
                    }
                    if(dataPoint.getRollAcceleration()>5){
                          roll+=1;
                          total+=1;
                    }
                    if(dataPoint.getYawAcceleration()>5){
                          yaw+=1;
                          total+=1;
                    }
            }

            dataset.addValue(total, series1, category1);
            dataset.addValue(pitch, series1, category2);
            dataset.addValue(roll, series1, category3);
            dataset.addValue(yaw, series1, category4);
            return dataset;
    }
}
```

Data Parser

```
package gui;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.TreeSet;

public class DataParser {
```

```
	public static TreeSet<Data> parseFile(String filePath){
		TreeSet<Data> data = new TreeSet<Data>();
		File file = new File(filePath);
		if(file.exists()){
			try {
				FileReader input = new FileReader(filePath);
				BufferedReader bufRead = new BufferedReader(input);
				String line = null;
				while((line=bufRead.readLine())!=null){
					String[] lineArray = line.split("\\,");
					data.add(new
Data(Double.parseDouble(lineArray[0]),
							Double.parseDouble(lineArray[1]),
							Double.parseDouble(lineArray[2]),
							Double.parseDouble(lineArray[3])));
				}
			} catch (FileNotFoundException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (IOException ie) {
				ie.printStackTrace();
			}
		}
		return data;
	}
}
```

Data Object

```
package gui;

public class Data implements Comparable<Data> {

	private Double time = null;
	private Double pitchAcceleration = null;
	private Double rollAcceleration = null;
	private Double yawAcceleration = null;

	public Data(Double time, Double pitchAcceleration, Double
rollAcceleration,
			Double yawAcceleration){
		this.time = time;
		this.pitchAcceleration = pitchAcceleration;
		this.rollAcceleration = rollAcceleration;
		this.yawAcceleration = yawAcceleration;
	}

	public int compareTo(Data data){
		if(this.time==data.time)
			return 0;
		if(this.time>data.time)
			return 1;
```

```
            if(this.time<data.time)
                  return -1;
            return 0;
      }

      public Double getTime() {
            return time;
      }

      public Double getPitchAcceleration() {
            return pitchAcceleration;
      }

      public Double getRollAcceleration() {
            return rollAcceleration;
      }

      public Double getYawAcceleration() {
            return yawAcceleration;
      }

}
```

Initial Screen XML

```
<panel name="panel" columns="2" gap="4" top="4" left="4">
  <label name="label1" text="Enter A Directory To View Flight Data FileNames"
/>
  <label name="label2"/>
  <textfield name="dir" text="Enter Directory Here" columns="30" />
  <button name="dirButton" text="Go" action="getFiles(dir.text)"/>
  <textarea name="fileList" wrap="true" columns="30" rows="5" rowspan="6"/>
  <label name="label3"/>
  <label name="label4"/>
  <label name="label5"/>
  <label name="label6"/>
  <label name="label7"/>
  <label name="label8"/>
  <label name="label9" text="Enter Absolute Path Of File To View" />
  <label name="label10"/>
  <textfield name="path" text="Enter Path Here" columns="30" />
  <button name="pathButton" text="Go" action="getData(path.text)"/>
</panel>
```

Final Screen XML

```
<panel name="panel" columns="2" gap="4" top="4" left="4">
  <button name="line" text="Total: Line Chart"
action="displayChart(line.text)"/>
```

```
  <button name="bar" text="Total: Bar Chart"
action="displayChart(bar.text)"/>
  <button name="pLine" text="Pitch: Line Chart"
action="displayChart(pLine.text)"/>
  <button name="rLine" text="Roll: Line Chart"
action="displayChart(rLine.text)"/>
  <button name="yLine" text="Yaw: Line Chart"
action="displayChart(yLine.text)"/>
  <label name="chart" valign="bottom" colspan="2" icon="lineChart.png" />
</panel>
```